UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

VINÍCIUS GARCIA PINTO

Escalonamento por Roubo de Tarefas em Sistemas Multi-CPU e Multi-GPU

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, março de 2013

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Pinto, Vinícius Garcia

Escalonamento por Roubo de Tarefas em Sistemas Multi-CPU e Multi-GPU / Vinícius Garcia Pinto. – Porto Alegre: PPGC da UFRGS, 2013.

87 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2013. Orientador: Nicolas Maillard.

1. Programação paralela. 2. Programação paralela híbrida. 3. GPU. 4. Ferramentas para programação paralela. 5. Escalonamento por roubo de tarefas. I. Maillard, Nicolas. II. Título.

“Um bom começo é a metade.”
— ARISTÓTELES

AGRADECIMENTOS

Agradeço aos meus pais Claive e Laerte, e a minha irmã Larissa pelo apoio e por acreditarem em mim. Agradeço à Franciele que sabe tudo o que representa para mim. Agradeço ao meu orientador Nicolas e aos colegas Bruno, Claudio, João, Julio, Silvio e Stéfano sem os quais este trabalho não seria possível. Agradeço aos demais colegas do GPPD pela cooperação acadêmica e pelas discussões proporcionadas. Agradeço ao Instituto de Informática da UFRGS e ao Centro Nacional de Supercomputação pelo apoio técnico e logístico para a realização deste trabalho. Por fim, agradeço as agências de fomento (Capes, CNPq e FAPERGS) por financiarem este trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ACML	AMD Core Math Library
API	Application Programming Interface
ATLAS	Automatically Tuned Linear Algebra Software
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
CESUP	Centro Nacional de Supercomputação
CUBLAS	CUDA Basic Linear Algebra Subprograms
CUDA	Compute Unified Device Architecture
CUFFT	CUDA Fast Fourier Transform
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
D&C	Divisão e Conquista
DEQUE	Double-Ended QUEue
DGEMM	Double-precision General Matrix Multiply
HEFT	Heterogeneous Earliest Finish Time
GCC	GNU Compiler Collection
GPPD	Grupo de Processamento Paralelo e Distribuído
GPU	Graphics Processing Unit
KAAPI	Kernel for Adaptative, Asynchronous Parallel and Interactive programming
MPI	Message-Passing Interface
OpenMP	Open Multi-Processing
PAD	Processamento de Alto Desempenho
PGAS	Partitioned Global Address Space
SINAPAD	Sistema Nacional de Processamento de Alto Desempenho
STL	Standard Template Library
TBB	Intel Threading Building Blocks
UFRGS	Universidade Federal do Rio Grande do Sul

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE TABELAS

RESUMO

Nos últimos anos, uma das alternativas adotadas para aumentar o desempenho de sistemas de processamento de alto desempenho têm sido o uso de arquiteturas híbridas. Essas arquiteturas são constituídas de processadores *multicore* e coprocessadores especializados, como GPUs. Esses coprocessadores atuam como aceleradores em alguns tipos de operações. Por outro lado, as ferramentas e modelos de programação paralela atuais não são adequados para cenários híbridos, produzindo aplicações pouco portáveis. O paralelismo de tarefas considerado um paradigma de programação genérico e de alto nível pode ser adotado neste cenário. Porém, exige o uso de algoritmos de escalonamento dinâmicos, como o algoritmo de roubo de tarefas.

Neste contexto, este trabalho apresenta um *middleware* (WORMS) que oferece suporte ao paralelismo de tarefas com escalonamento por roubo de tarefas em sistemas híbridos multi-CPU e multi-GPU. Esse *middleware* permite que as tarefas tenham implementação tanto para execução em CPUs quanto em GPUs, decidindo em tempo de execução qual das implementações será executada de acordo com os recursos de *hardware* disponíveis.

Os resultados obtidos com o WORMS mostram ser possível superar, em algumas aplicações, tanto o desempenho de ferramentas de referência para execução em CPU quanto de ferramentas para execução em GPUs.

Palavras-chave: Programação paralela, programação paralela híbrida, GPU, ferramentas para programação paralela, escalonamento por roubo de tarefas.

ABSTRACT

In the last years, one of alternatives adopted to increase performance in high performance computing systems have been the use of hybrid architectures. These architectures consist of multicore processors and specialized coprocessors, like GPUs. Coprocessors act as accelerators in some types of operations. On the other hand, current parallel programming models and tools are not suitable for hybrid scenarios, generating less portable applications. Task parallelism, considered a generic and high level programming paradigm, can be used in this scenario. However, it requires the use of dynamic scheduling algorithms, such as work stealing.

In this context, this work presents a middleware (WORMS) that supports task parallelism with work stealing scheduling in multi-CPU and multi-GPU systems. This middleware allows task implementations for both CPU and GPU, deciding at runtime which implementation will run according to the available hardware resources.

The performance results obtained with WORMS showed that is possible to outperform both CPU and GPU reference tools in some applications.

Keywords: parallel programming, hybrid parallel programming, GPU, parallel programming tools, work stealing scheduling.

1 INTRODUÇÃO

Até meados da última década o aumento de desempenho das arquiteturas de computadores esteve relacionado à diminuição do tamanho dos transistores e na elevação da frequência de *clock* do processador. Durante esse período, a cada ano os computadores sequenciais ficavam em média, de 40 a 50% mais rápidos que os modelos do ano anterior (LARUS, 2009), o que permitiu que o *software* executasse mais rápido no novo *hardware* com pouca ou nenhuma alteração no código fonte (CALLAHAN, 2008; BUTTARI et al., 2007). Entretanto, esse modelo baseado no processador tradicional tornou-se inviável, pois atingiu limites físicos em aspectos como a dissipação de calor, o consumo de energia e a diferença entre desempenho do processador e o desempenho do sistema de memória (BORKAR; CHIEN, 2011).

A partir de então, novas formas têm sido adotadas para melhorar o desempenho como o uso de processadores *multicore*, constituídos por dois ou mais núcleos de processamento encapsulados em um mesmo *chip* e a construção de sistemas híbridos, que combinam processadores de propósito geral com coprocessadores especializados (BUTTARI et al., 2007; ASANOVIC et al., 2009). Os coprocessadores especializados possuem desempenho otimizado para determinado tipo de operação, como as GPUs (*Graphics Processing Units*), que são empregadas como aceleradores no cálculo de tarefas paralelas com poucas dependências, ainda que originalmente tenham sido projetadas somente para o processamento gráfico (TOMOV; DONGARRA; BABOULIN, 2010).

Os sistemas de processamento de alto desempenho (PAD) até então predominantemente limitados a arquiteturas multiprocessador, máquinas vetoriais e agregados (NAVAUX; ROSE, 2008), passaram também a serem construídos de forma híbrida, incorporando em suas arquiteturas o uso de processadores *multicore* e de aceleradores como GPUs (KISTLER et al., 2009). Essa tendência pode ser observada pelo crescimento da utilização de aceleradores e coprocessadores nos sistemas classificados na lista TOP 500[1] (TOP500, 2012), conforme mostrado na Tabela 1.1. Na edição de Novembro de 2012, dos 500 computadores classificados pela lista, 62 fazem uso de aceleradores em suas arquiteturas. Como exemplo, pode-se citar o primeiro colocado no TOP 500, o computador *Titan* cuja arquitetura é composta por 18688 nós, cada um composto por um processador AMD Opteron 6274 (16 *cores*) e por uma GPU NVIDIA Tesla K20. Conforme pode ser visto na Tabela 1.2, entre os 62 computadores classificados pelo TOP 500 que utilizam algum tipo de acelerador, 53 utilizam GPUs (TOP500, 2012).

Esse novo contexto no qual os sistemas de computação de alto desempenho estão inseridos mostra que os modelos e ferramentas atuais para programação paralela não são adequados para explorar o potencial de computação existente em um cenário composto por

[1] A lista TOP500 classifica os 500 computadores com maior capacidade de processamento no mundo.

Edição	Sistemas com Aceleradores	%
11/2012	62	12,4
06/2012	57	11,4
11/2011	39	7,8
06/2011	19	3,8
11/2010	17	3,4
06/2010	9	1,8
11/2009	7	1,4
06/2009	5	1,0
11/2008	8	1,6
06/2008	4	0,8
11/2007	1	0,2
06/2007	1	0,2
11/2006	1	0,2
06/2006	1	0,2
11/2005	0	0,0

Tabela 1.1: Evolução do uso de aceleradores/coprocessadores nos computadores listados no TOP500.

múltiplos e distintos núcleos de processamento (VANDIERENDONCK; PRATIKAKIS; NIKOLOPOULOS, 2011; PEREZ; BADIA; LABARTA, 2008; BUTTARI et al., 2007). Isso ocorre porque esses modelos exigem esforços do programador para determinar tanto as operações que são executadas em paralelo por cada processador quanto os pontos de bloqueio e sincronização necessários. A opção mais frequente nesses sistemas híbridos é programar cada recurso de *hardware* com sua API específica, tornando os programas menos legíveis e pouco portáveis.

O paralelismo de tarefas, implementado por ferramentas de programação paralela como Cilk (BLUMOFE et al., 1995), OpenMP 3 (OpenMP ARB, 2008) e Intel *Threading Building Blocks* (TBB) (REINDERS, 2010), é considerado um paradigma de programação adequado para as demandas existentes nesse contexto por ser genérico e expressivo (KAMBADUR et al., 2009). Com o paralelismo de tarefas pode-se descrever o algoritmo em função de tarefas, permitindo, em um primeiro momento, abstrair os diferentes recursos de processamento do sistema.

Apesar de ser adequado, para oferecer desempenho satisfatório o paralelismo de tarefas demanda por técnicas de escalonamento e balanceamento de carga eficientes que propiciem o máximo aproveitamento dos recursos de processamento disponíveis. Em sistemas híbridos essas demandas são acentuadas devido à capacidade de processamento distinta entre os recursos, que podem provocar ou acentuar o desbalanceamento da carga de trabalho. A técnica de escalonamento por roubo de tarefas (*work stealing*) têm sido utilizada no escalonamento de tarefas paralelas em ferramentas para programação em CPUs *multicore* como Cilk (BLUMOFE et al., 1995) e Intel TBB (REINDERS, 2010). No contexto do processamento em GPUs, o roubo de tarefas tem sido adotado em trabalhos recentes tanto em cenários multi-GPUs (LIMA et al., 2012) quanto no cenário intra-GPU (TOSS; GAUTIER, 2012; CEDERMAN; TSIGAS, 2011, 2008).

Acelerador	Número	%
NVIDIA 2090	30	48,38
NVIDIA 2050	11	17,74
NVIDIA 2070	7	11,29
Intel Xeon Phi	5	8,06
NVIDIA K20x	2	3,22
IBM PowerXCell 8i	2	3,22
Intel Xeon Phi 5110P	2	3,22
AMD Radeon HD 7970	1	1,61
AMD FirePro S10000	1	1,61
ATI GPU	1	1,61
Total GPUs	53	85,48
Total	62	100,00

Tabela 1.2: Uso de aceleradores nos computadores listados no TOP500 em Novembro de 2012.

1.1 Proposta deste Trabalho

Este trabalho tem por objetivo explorar e validar o escalonamento baseado na estratégia de roubo de tarefas em sistemas paralelos híbridos com suporte simultâneo a recursos de processamento heterogêneos como CPUs *multicore* e múltiplas GPUs. Esta proposta foi implementada em um protótipo chamado WORMS (WORk stealing scheduling for Multi-CPU/GPU Systems) que possibilita a execução de tarefas com dependências estritas em CPUs e GPUs simultaneamente escalonando as tarefas por meio da técnica de roubo de tarefas. No WORMS as tarefas possuem implementações tanto para execução em CPU quanto em GPU, sendo executadas em recursos computacionais virtuais identificados como *TaskProcessors*. Os *TaskProcessors* podem representar tanto os *cores* de uma CPU *multicore* quanto as placas GPU instaladas no sistema.

Com isso, visa-se obter um aproveitamento mais eficiente do potencial de paralelismo dos recursos existentes na arquitetura computacional.

1.2 Organização do Texto

O restante desta dissertação está organizada da seguinte forma: o Capítulo 2 apresenta os conceitos de paralelismo de tarefas e de escalonamento por roubo de tarefas. Esse capítulo aborda também as ferramentas que oferecem suporte ao paralelismo de tarefas e alguns exemplos de algoritmos que podem ser descritos com este paradigma.

O Capítulo 3 apresenta a proposta deste trabalho para oferecer suporte ao paralelismo de tarefas com escalonamento por roubo de tarefas em arquiteturas híbridas Multi-CPU e Multi-GPU. Adicionalmente, o Capítulo 3 discute as versões, o funcionamento, a implementação, a API e as limitações da proposta apresentada.

O Capítulo 4 aborda a avaliação experimental da proposta apresentada no Capítulo 3 bem como a avaliação das ferramentas de referência utilizadas como parâmetros de comparação. Essa avaliação experimental compara os resultados obtidos neste trabalho com os resultados apresentados pelas ferramentas de referência.

Por fim, o Capítulo 5 apresenta as conclusões, contribuições e trabalhos futuros desta dissertação. Adicionalmente, o Apêndice A apresenta alguns experimentos complementares realizados durante o desenvolvimento deste trabalho.

2 PARALELISMO DE TAREFAS E ESCALONAMENTO POR ROUBO DE TAREFAS

Este capítulo apresenta o paralelismo de tarefas e o escalonamento por roubo de tarefas. Além destes conceitos também são abordadas algumas ferramentas que oferecem suporte ao paralelismo de tarefas e exemplos de algoritmos que podem ser descritos nesse paradigma.

2.1 Paralelismo de Tarefas

A computação de muitos problemas sequenciais contém partes que são independentes entre si e que podem ser executadas em paralelo. Essas partes, que podem ser chamadas de tarefas, consistem em unidades de computação com operações sequenciais (p.ex. uma função ou uma sequência de instruções) definidas pelo programador (RAUBER; RÜNGER, 2010).

As tarefas nas quais a computação de um problema é subdividida são obtidas através de métodos de decomposição. As múltiplas tarefas oriundas da decomposição podem não ser do mesmo tamanho e a execução simultânea dessas é a chave para reduzir o tempo necessário para resolver o problema inteiro. Além disso, as tarefas podem ter dependências entre si ou podem ser completamente independentes (GRAMA, 2003; MATTSON; SANDERS; MASSINGILL, 2005).

O paralelismo de tarefas (*task parallelism*) é o tipo de paralelismo que é naturalmente expresso por meio de tarefas paralelas em um grafo de dependência de tarefas. Algoritmos como o *quicksort* paralelo, a fatoração de matrizes esparsas e os algoritmos derivados pela decomposição por divisão e conquista são exemplos de algoritmos que podem ser descritos nesse paradigma (GRAMA, 2003).

O grafo de dependência de tarefas é um grafo direcionado acíclico ou DAG (*Directed Acyclic Graph*) utilizado para representar as tarefas e suas dependências. Nesse grafo, os nós representam as tarefas a serem executadas e as arestas representam as dependências e as precedências entre elas. A Figura 2.1 apresenta um exemplo de grafo de dependência de tarefas para o algoritmo de ordenação mergesort. Nesse exemplo, as tarefas T_1, T_2 e T_3 são tarefas de subdivisão, que dividem a entrada atual em duas novas tarefas, cada uma com metade do tamanho de entrada inicial, as tarefas T_4, T_5, T_6 e T_7 ordenam sequencialmente a entrada recebida enquanto as tarefas T_8, T_9 e T_{10} fazem a combinação de duas entradas previamente ordenadas. Dessa forma, conforme ilustrado no grafo, as tarefas de mistura não podem ser executadas antes que as tarefas de subdivisão e ordenação tenham sido concluídas, assim como as tarefas de ordenação não podem ser processadas antes que as tarefas de subdivisão relacionadas tenham sido executadas.

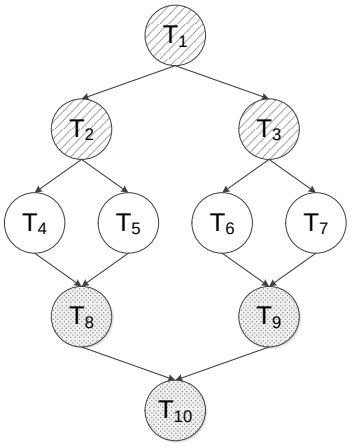


Figura 2.1: Exemplo de um grafo de dependência de tarefas para o algoritmo mergesort.

Cada subtarefa obtida de um problema inteiro a partir de algum método de decomposição tem uma quantidade de trabalho sequencial associado a ela. Essa quantidade de trabalho é chamada de granularidade ou tamanho da tarefa. Em geral, um problema pode ser subdivido em tarefas de diferentes tamanhos, porém essa decomposição deve atender a dois critérios conflitantes:

- o número de tarefas deve ser o maior possível para oferecer flexibilidade ao escalonador, ou seja, as tarefas devem ter granularidade pequena (VALIANT, 1990; FRIGO; LEISERSON; RANDALL, 1998);

- a granularidade associada a cada tarefa deve ser grande o suficiente para minimizar os *overheads* de gerenciamento e tratamento das dependências das tarefas (MATTSON; SANDERS; MASSINGILL, 2005).

Explorar o potencial de paralelismo de um problema utilizando paralelismo de tarefas torna necessário, na maioria das vezes, o uso de algoritmos de escalonamento complexos (MATTSON; SANDERS; MASSINGILL, 2005). Em alguns problemas todas as tarefas são conhecidas no inicio da computação, enquanto em outros uma tarefa pode dar origem a novas sub-tarefas de forma dinâmica em tempo de execução. Além disso, as novas tarefas podem ou não possuir dependências com as demais tarefas (MOR, 2010). Dessa forma, distribuir adequadamente as tarefas entre os recursos de processamento, e redistribuí-las, caso necessário, durante a computação é um aspecto essencial para evitar desbalanceamento de carga e permitir que o programa execute de forma mais eficiente.

Nesse contexto, são utilizadas duas classes de algoritmos de escalonamento: os escalonadores estáticos, onde o mapeamento das tarefas aos recursos de processamento é determinado em tempo de compilação ou no início da execução e não varia durante a computação; e os escalonadores dinâmicos, nos quais a distribuição das tarefas entre esses recursos varia conforme decorre a computação.

Nos escalonadores estáticos o mapeamento das tarefas aos recursos pode ser baseado em estimativas dos tempos de execução das tarefas obtidos a partir de resultados experimentais ou por uma análise da estrutura computacional das tarefas (RAUBER; RÜNGER, 2010). São usados quando a granularidade associada às tarefas e a capacidade dos recursos que compõem o sistema são estáveis e/ou previsíveis (MATTSON; SANDERS; MASSINGILL, 2005).

Já os escalonadores dinâmicos são usados, em geral, quando a quantidade de trabalho associada a cada tarefa varia muito ou é imprevisível e/ou quando a capacidade dos recursos de processamento do sistema varia ou é imprevisível (MATTSON; SANDERS; MASSINGILL, 2005). Escalonadores dinâmicos são usualmente classificados em abordagens centralizadas ou distribuídas (GRAMA, 2003).

Uma estratégia simples de escalonamento dinâmico é a abordagem centralizada que consiste em manter uma fila global de tarefas que é usada por todos os recursos de processamento. Sempre que um recurso está sem tarefas para processar, ele busca uma nova tarefa da fila centralizada e passa a executá-la. Os recursos de processamento mais rápidos, ou que receberem tarefas menores acessam a fila com maior frequência e consequentemente processam mais tarefas. Entretanto, manter uma única fila global, compartilhada entre todos os recursos de processamento, implica em *overheads* nas buscas e inserções de tarefas (MATTSON; SANDERS; MASSINGILL, 2005) o que torna limitada a escalabilidade dessa estratégia (GRAMA, 2003).

Outras estratégias de escalonamento dinâmico que não implicam no uso de uma fila centralizada são as abordagens distribuídas como o escalonamento por roubo de tarefas (*work stealing*) e o *work pushing/sharing* (MATTSON; SANDERS; MASSINGILL, 2005). Essas abordagens estão menos sujeitas aos gargalos associados com as estratégias centralizadas, pois cada recurso de processamento tem sua própria fila de tarefas (GRAMA, 2003). Dessa forma, um recurso pode enviar e/ou buscar tarefas de outros recursos. No *work pushing*, também referenciado como *work sharing*, sempre que um recurso de processamento cria uma nova tarefa, o escalonador procura um recurso para o qual enviará a nova tarefa. Já no roubo de tarefas, os recursos de processamento subutilizados é que buscam tarefas em outros recursos (BLUMOFE; LEISERSON, 1994).

2.2 Escalonamento por Roubo de Tarefas

O escalonamento por roubo de tarefas (*work stealing*) é um algoritmo de escalonamento que utiliza filas distribuídas para escalonar tarefas entre os processadores ou unidades de processamento. Nessa técnica, cada unidade de processamento possui uma fila de tarefas a processar. A fila de tarefas de cada unidade pode ser acessada tanto pela própria unidade quanto por outras unidades. Quando uma unidade de processamento se torna ociosa, ou seja, está com a sua lista de tarefas a executar vazia, escolhe uma outra unidade de processamento da qual roubará tarefas e então, caso essa unidade tenha tarefas a executar na lista, rouba a última tarefa inserida na lista de execução desta unidade. A unidade de processamento ociosa que realiza os roubos é identificada como ladrão, enquanto a unidade da qual as tarefas são roubadas é identificada como vítima. Um dos aspectos principais do escalonamento por roubo de tarefas é que o custo do escalonamento ou da decisão de onde uma tarefa será executada é transferido de um recurso de processamento que está ocupado para um recurso de processamento que está ocioso. Essa abordagem contrasta com outras estratégias como *work pushing* onde o recurso de processamento que cria a tarefa é que decide onde ela será executada.

Em (BLUMOFE; LEISERSON, 1994) é proposto um escalonador baseado em roubo de tarefas (*work stealing*), no qual a escolha da vítima da qual as tarefas serão roubadas é feita por meio de uma estratégia aleatória. Esse escalonador é a base para a ferramenta de programação paralela Cilk (BLUMOFE et al., 1995; FRIGO; LEISERSON; RANDALL, 1998).

O escalonamento por roubo de tarefas é utilizado como técnica de escalonamento efi-

ciente por diversas ferramentas para programação paralela como Cilk (BLUMOFE et al., 1995) e Intel TBB (REINDERS, 2010). Na literatura podem ser encontradas várias propostas de alterações na abordagem original a fim de adequar a técnica a cenários específicos:

- O ***adaptive work stealing*** (AGRAWAL et al., 2008) possibilita oferecer melhor desempenho em sistemas com um grande número de processadores onde muitos *jobs* podem compartilhar o mesmo recurso de processamento ou quando o número de processadores alocado a um determinado *job* pode variar durante a execução do mesmo;

- No ***idempotent work stealing*** (MICHAEL; VECHEV; SARASWAT, 2009) as alterações introduzidas permitem que uma tarefa possa ser executada mais de uma vez. Dessa forma as operações de bloqueio e sincronização são reduzidas o que possibilita desempenho superior à versão original em algumas classes de aplicações;

- O ***scalable work stealing*** (DINAN et al., 2009) possibilita a utilização eficiente do escalonamento por roubo de tarefas em sistemas de memória distribuída de larga escala que utilizam PGAS (*Partitioned Global Address Space*);

- Os algoritmos **RATMD e RTMPD** (MOR; MAILLARD, 2011; MOR, 2010) possibilitam a utilização do escalonamento baseado em roubo de tarefas em aplicações que utilizam a interface de troca de mensagens MPI (FORUM, 1997);

- O ***enhanced cilk scheduler*** (BENDER; RABIN, 2000) projeta modificações no escalonador do Cilk visando oferecer suporte a sistemas onde os processadores possuem capacidades de processamento e custos de comunicação diferentes entre si.

O escalonamento por roubo de tarefas também têm sido avaliado no contexto do processamento em GPUs para proporcionar balanceamento de carga entre os *streaming multiprocessors* que compõe estes recursos. As abordagens implementadas em (CEDERMAN; TSIGAS, 2008, 2011; TOSS; GAUTIER, 2012) mostram a aplicabilidade do roubo de tarefas nesse cenário e a melhora no desempenho em comparação com as abordagens com uso de fila dinâmica bloqueante, fila dinâmica não-bloqueante e fila estática, esta última comumente utilizada no escalonamento e balanceamento de carga intra-GPUs.

2.3 Suporte ao Paralelismo de Tarefas

Diversas ferramentas de programação suportam paralelismo de tarefas em CPUs *multicore*. Recentemente, alguns trabalhos em andamento têm buscado oferecer suporte a esse paradigma também em sistemas híbridos compostos por CPUs e GPUs.

2.3.1 Cilk

A ferramenta para programação paralela Cilk (BLUMOFE et al., 1995; FRIGO; LEISERSON; RANDALL, 1998; GROUP, 2012) é uma extensão da linguagem C para permitir a submissão, execução e sincronização de tarefas paralelas. A implementação de Cilk é baseada no algoritmo de escalonamento dinâmico por meio de roubo de tarefas apresentado em (BLUMOFE; LEISERSON, 1994). Cilk define tarefas como funções individuais que podem submeter novas tarefas dinamicamente. A sincronização é feita

permitindo que as tarefas esperem pelas tarefas filhas, ou seja, tarefas que foram submetidas pela tarefa original.

O Código 2.1 ilustra um exemplo de código em Cilk. A palavra chave *spawn* indica que a função a ser chamada será submetida como uma nova tarefa. A palavra chave *sync* indica que uma tarefa deve esperar por todas as suas tarefas filhas. As palavras chave *spawn* e *sync* podem ser utilizadas somente em funções definidas com a palavra chave *cilk*.

Código 2.1: Exemplo de código Cilk para calcular o enésimo número de Fibonacci em paralelo.

```
cilk int fib(int n){
   if (n < 2)
      return (n);
   else {
      int x, y;
      x = spawn fib(n - 1);
      y = spawn fib(n - 2);
      sync;
      return (x + y);
   }
}

cilk int main(int argc, char *argv[]){
   int n, result;
   n = atoi(argv[1]);
   result = spawn fib(n);
   sync;
   printf("Result: %d\n", result);
   return 0;
}
```

2.3.2 Intel TBB

O Intel TBB (*Threading Building Blocks*) (REINDERS, 2010; CORPORATION, 2012) é uma biblioteca baseada em templates para programação paralela em C++ que faz uso de *threads*. Essa biblioteca permite expressar o paralelismo em diversos paradigmas de programação paralela como o paralelismo de dados, de laços ou de tarefas. As unidades de trabalho paralelas resultantes são escalonadas em *threads* por meio de um algoritmo de roubo de tarefas inspirado no ambiente Cilk.

O Código 2.2 apresenta um exemplo de código com Intel TBB para efetuar o cálculo em paralelo do enésimo número da sequência de Fibonacci.

Código 2.2: Exemplo de código Intel TBB para calcular o enésimo número de Fibonacci.

```
class FibTask: public task {
public:
   const long n;
   long* const sum;
   FibTask( long n_, long* sum_ ) : n(n_), sum(sum_){}
   task* execute( ) {
      if( n < CutOff ) {
         *sum = SerialFib(n);
      } else {
         long x, y;
         FibTask& a = *new( allocate_child( ) ) FibTask(n-1,&x);
         FibTask& b = *new( allocate_child( ) ) FibTask(n-2,&y);
```

```
            set_ref_count(3);
            spawn( b );
            spawn_and_wait_for_all( a );
            *sum = x+y;
        }
        return NULL;
    }
};
```

2.3.3 OpenMP

OpenMP (*Open Multi-Processing*) (CHAPMAN; JOST; PAS, 2007) é uma ferramenta para programação paralela baseada na adição de diretivas de compilação em códigos C, C++ e Fortran. As diretivas OpenMP são utilizadas para indicar a paralelização de laços ou trechos de código. A partir da versão 3.0 (OpenMP ARB, 2008) foi inserido o conceito de tarefas, o que permite a utilização do paralelismo de tarefas através do uso de diretivas para delimitação de trechos de código como unidades de trabalho paralelas (AYGUADE et al., 2009). A especificação OpenMP (OpenMP ARB, 2008) não define uma política para o escalonamento das tarefas, ficando essa escolha a cargo de cada implementação.

Código 2.3: Exemplo de código OpenMP task.

```
void mergeSort(TYPE vec[], int vecSize, int threshold) {
    int mid;
    if (vecSize > threshold) {
        mid = vecSize / 2;
        #pragma omp task
            mergeSort(vec, mid, threshold);
        #pragma omp task
            mergeSort(vec + mid, vecSize - mid, threshold);
        #pragma omp taskwait
        #pragma omp task
            merge(vec, vecSize);
        #pragma omp taskwait
    } else
        mergeSortSeq(vec, vecSize);
}

int main(int argc, char *argv[]){
    TYPE *vec = (TYPE*) malloc(SIZE * sizeof (TYPE));
    for ( int i = 0; i < SIZE; i++){
        vec[tmp] = rand() % 99;
    }
    #pragma omp parallel
    #pragma omp single
        mergeSort(vec, SIZE, THRESHOLD);
    return 0;
}
```

2.3.4 XKaapi

O XKaapi (INRIA; MOAIS; LIG, 2011) é uma reimplementação do KAAPI com suporte a paralelismo de tarefas de grão fino. O KAAPI (GAUTIER; BESSERON; PIGEON, 2007) (*Kernel for Adaptative, Asynchronous Parallel and Interactive programming*) é uma ferramenta para computação paralela em CPUs *multicore* e *clusters*. A implementação atual do XKaapi oferece suporte a arquiteturas *multicore* e extensões para

suporte eficiente à GPUs foram propostas em (HERMANN et al., 2010; LIMA et al., 2012). O XKaapi é composto por um conjunto de APIs (*Application Programming Interfaces*) e pelo *kernel*, um ambiente de execução para as APIs que oferece escalonamento baseado em roubo de tarefas. A Kaapi++ é a interface do XKaapi baseada em um DFG (*Data Flow Graph*) para C++ e é dividida em três componentes: a assinatura da tarefa (*task signature*) onde são definidos o número e as características dos parâmetros da tarefa; a implementação da tarefa (*task implementation)* que especifica a implementação da tarefa para uma arquitetura e a criação da tarefa (*task creation*) que submete a tarefa para a pilha de execução.

O Código 2.4 apresenta um exemplo de tarefa descrita em Kaapi++, nesse código existem implementações CPU e GPU para uma mesma tarefa conforme proposto em (HERMANN et al., 2010), dessa forma o escalonador do XKaapi decide em tempo de execução qual das implementações será utilizada.

Código 2.4: Exemplo de código para definição de tarefa em XKaapi/Kaapi++. (LIMA et al., 2012)

```
struct TaskSYRK: public ka::Task<2>::Signature< ka::R <ka::range2d<
    double>>, ka::RW <ka::range2d<double>> > {};

template<> struct TaskBodyCPU<TaskSYRK>{
    void operator ( ka::range2d_r<double> A, ka::range2d_rw<double> C ){
        /* CPU implementation */
    }
};

template<> struct TaskBodyGPU<TaskSYRK>{
    void operator ( ka::gpuStream stream, ka::range2d_r<double> A, ka::
        range2d_rw<double> C ){
        /* GPU implementation */
    }
};
```

2.3.5 StarPU

O StarPU (AUGONNET et al., 2009) é uma ferramenta para programação paralela que oferece suporte para arquiteturas híbridas, como CPUs *multicore* e aceleradores. O StarPU propõe uma abordagem de tarefas independente da arquitetura base. São definidos *codelets* como uma abstração de uma tarefa que pode ser executada em um núcleo de uma CPU *multicore* ou submetido a um acelerador. Cada *codelet* pode ter múltiplas implementações, uma para cada arquitetura em que o *codelet* pode ser executado, utilizando as linguagens ou bibliotecas específicas para a arquitetura alvo. Uma aplicação StarPU é descrita como um conjunto de *codelets* com suas dependências de dados (AUGONNET; NAMYST, 2009).

A ferramenta possui um conjunto de políticas de escalonamento implementadas que o programador pode escolher de acordo com as características da aplicação. A principal delas faz uso do algoritmo de escalonamento estático HEFT (*Heterogeneous Earliest Finish Time*) para escalonar as tarefas com base em modelos de custo de execução das tarefas.

O Código 2.5 ilustra um exemplo de código para declaração e implementações de um *codelet* bem como da submissão de uma tarefa a partir desse *codelet*.

Código 2.5: Exemplo de código para definição e submissão de tarefa em StarPU.

```
void scal_gpu(void *buffers[], void *cl_arg){
```

```
    /* CUDA code */
}
void scal_cpu(void *buffers[], void *cl_arg){
    /* CPU code */
}

static starpu_codelet scal_cl = {
     .where = STARPU_CPU|STARPU_CUDA,
     .cpu_func = scal_cpu,
     .cuda_func = scal_gpu,
     .model = &starpu_scal_model,
     .nbuffers = N_BUFFERS
}

...
struct starpu_task *task = starpu_task_create();
...
task->cl = &scal_cl;
task->buffers[0].handle = vector_handle;
task->buffers[0].mode = STARPU_RW;
task->cl_arg = &factor;
task->cl_arg_size = sizeof(factor);
...
starpu_task_submit(task);
starpu_task_wait_for_all();
...
```

2.4 Exemplos de Algoritmos Descritos com Paralelismo de Tarefas

Nesta seção são apresentados exemplos de alguns algoritmos que podem ser descritos com paralelismo de tarefas. Esses algoritmos são derivados pela decomposição por divisão e conquista (D&C) o que possibilita alto potencial de concorrência pois os subproblemas gerados são independentes e podem ser computados paralelamente. Conforme pode ser visto na Figura 2.2 a cada etapa de subdivisão o potencial de paralelismo dobra. Entretanto, a partir de um certo nível de recursão, a quantidade de computação dos subproblemas torna-se muito pequena, de forma a não compensar a criação de novas tarefas. Nesse caso, alguns algoritmos passam a aplicar uma solução sequencial quando o tamanho do subproblema torna-se menor que um determinado limiar (threshold).

2.4.1 i-ésimo número de *Fibonacci*

Os números de Fibonacci são uma sequência de números inteiros onde cada elemento é a soma dos dois elementos anteriores, com exceção dos dois primeiros elementos que, por definição, são 0 e 1. Matematicamente a sequência é definida pela relação de recorrência $F_0 = 0$ se $n = 0$, $F_1 = 1$ se $n = 1$ e $F_n = F_{n-1} + F_{n-2}$ para $n \geq 2$. Dessa forma, obtém-se a sequencia numérica $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...$

Uma das formas para calcular o i-ésimo número da sequencia de fibonacci é por meio do algoritmo recursivo:

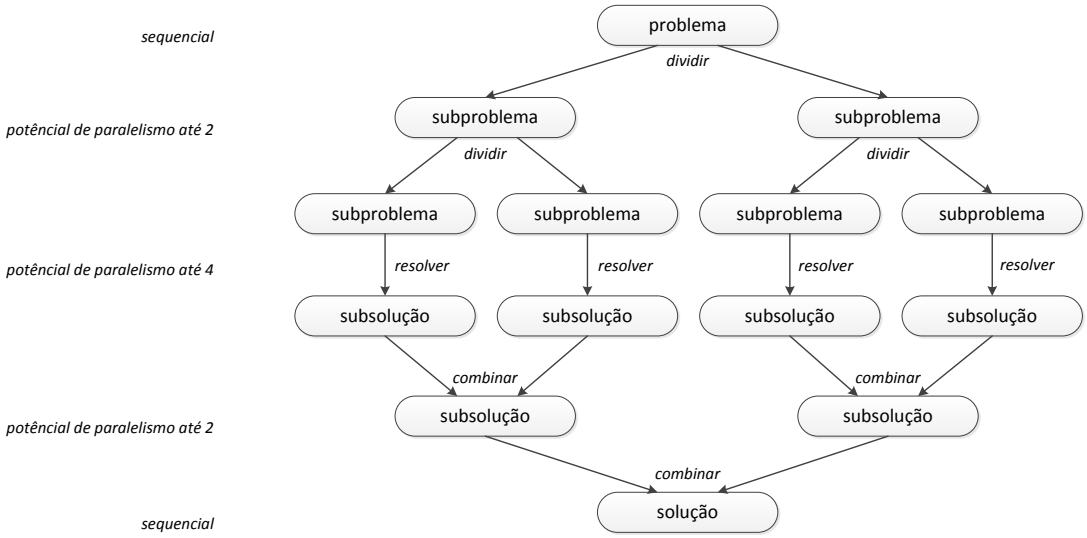


Figura 2.2: Potencial de paralelismo com Divisão & Conquista.

$\textsc{Fib}(n)$
  if $n < 2$ **then**
    return n
  else
    $x \leftarrow \textsc{Fib}(n-1)$
    $y \leftarrow \textsc{Fib}(n-2)$
    return $x + y$

Uma versão paralela deste algoritmo utilizando a ferramenta Cilk pode ser vista no código 2.1.

2.4.2 Transformação

Uma transformação é um algoritmo simples, semelhante a um *map*[1] de linguagens funcionais, que aplica uma operação a cada elemento de um vetor de entrada e então grava os resultados em um vetor de saída. A figura 2.3 ilustra o comportamento de um algoritmo de transformação que aplica uma operação de negação sobre um vetor de entrada.

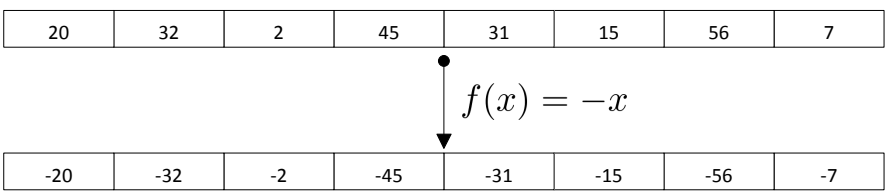


Figura 2.3: Exemplo da aplicação de uma transformação $f(x) = -x$ em um vetor.

[1]Em linguagens funcionais, um *map* é uma função de ordem superior que recebe como parâmetros uma lista de elementos e uma outra função. A função recebida é aplicada a cada elemento da lista gerando uma nova lista com os resultados (WATT, 2004; SEBESTA, 2003).

Um algoritmo de transformação é tradicionalmente escrito de forma iterativa como no Código 2.6, mas também pode ser descrito de forma recursiva utilizando tarefas, conforme é apresentado no Código 2.7.

Código 2.6: Exemplo de código para calcular uma transformação de forma iterativa.

```
void transform(TYPE* in, TYPE* out, int size, TYPE (*pfunc)(TYPE)){
   int i = 0;
   #pragma omp parallel for
   for(i=0 ; i<size; i++){
      out[i] = pfunc(in[i]);
   }
}
```

Código 2.7: Exemplo de código para calcular uma transformação de forma recursiva.

```
void transform(TYPE* in, TYPE* out, int size, TYPE (*pfunc)(TYPE)){
   if(size > 1){
      int mid = size / 2;
      #pragma omp task
         transform(in, out, mid, pfunc);
      #pragma omp task
         transform(in + mid, out + mid, size - mid, pfunc);
      #pragma omp taskwait
   }else
      out[0] = pfunc(in[0]);
}
```

2.4.3 Ordenação Mergesort

Um algoritmo de ordenação coloca os elementos de um vetor de entrada em uma ordem predefinida e então grava o resultado em um vetor de saída. O algoritmo de ordenação *mergesort* possui três etapas:

1. Dividir o vetor de entrada a ser ordenado em dois subvetores menores, cada um com tamanho $n/2$;

2. Ordenar os dois subvetores recursivamente utilizando mergesort;

3. Combinar os dois subvetores ordenados para formar um vetor ordenado do tamanho original.

Na versão clássica do algoritmo mergesort, o critério de parada das chamadas recursivas é quando a entrada a ser ordenada atinge o tamanho um, pois por definição, toda sequência de um elemento já está ordenada. Entretanto, para tamanhos suficientemente pequenos, outros algoritmos de ordenação, como o insertion sort, tendem a ser mais rápidos que o mergesort. Dessa forma, implementações atuais do mergesort utilizam tamanhos de entrada maiores como critério de parada para recursão e então aplicam algum algoritmo de ordenação mais rápido sobre o subvetor obtido. O algoritmo empregado nessas implementações é frequentemente chamado de mergesort híbrido.

O código 2.3 ilustra exemplo de uma implementação paralela do algoritmo mergesort utilizando as diretivas de tarefas da ferramenta OpenMP.

2.4.4 Multiplicação de Matrizes por Strassen

A multiplicação de duas matrizes quadradas A e B de ordem n x n que resulta na matriz C pode ser feita por meio do algoritmo obtido diretamente da definição desse problema, calculando cada elemento da matriz C da seguinte forma:

$$c_{i,j} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}, \qquad para\ 1 \leq i, j \leq n.$$

Outra forma de multiplicar duas matrizes quadradas é por meio do algoritmo de multiplicação de matrizes recursivo, porém, nesse caso as matrizes devem ser de ordem 2^n x 2^n. Esse algoritmo consiste em dividir recursivamente as matrizes em quatro blocos de $n/2$ x $n/2$, até que o tamanho das matrizes seja 1. Assim, cada bloco da matriz C é calculado a partir da soma de duas multiplicações de blocos das matrizes A e B. Entretanto, essa técnica envolve o mesmo número de multiplicações que o método obtido da definição.

O algoritmo de Strassen (NEAPOLITAN; NAIMIPOUR, 2010; CORMEN et al., 2009) é um algoritmo recursivo para multiplicação de matrizes que se baseia no método por blocos. No caso de uma matriz 2 x 2, com o uso de sete matrizes auxiliares, o método de Strassen permite diminuir o número de multiplicações necessário em comparação ao algoritmo da definição, de oito para sete, ainda que o número de adições e subtrações seja maior. As matrizes auxiliares e os blocos da matriz C são calculados da seguinte forma:

$$\begin{aligned}
M_1 &\leftarrow (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
M_2 &\leftarrow (A_{21} + A_{22}) \cdot B_{11} \\
M_3 &\leftarrow A_{11} \cdot (B_{12} - B_{22}) \\
M_4 &\leftarrow A_{22} \cdot (B_{21}\ B_{11}) \\
M_5 &\leftarrow (A_{11} + A_{12}) \cdot B_{22} \\
M_6 &\leftarrow (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
M_7 &\leftarrow (A_{12} - A_{22}) \cdot (B_{21} + B_{22})
\end{aligned}$$

$$\begin{aligned}
C_{11} &\leftarrow M_1 + M_4 - M_5 + M_7 \\
C_{12} &\leftarrow M_3 + M_5 \\
C_{21} &\leftarrow M_2 + M_4 \\
C_{22} &\leftarrow M_1 + M_3 - M_2 + M_6.
\end{aligned}$$

Na prática, implementações do método de Strassen utilizam um limiar (threshold) para o tamanho das submatrizes a serem multiplicadas, a partir do qual passa a ser aplicado um algoritmo de multiplicação de matrizes mais simples, como o método derivado da definição (CORMEN et al., 2009). Isso é feito porque o algoritmo recursivo de Strassen só é eficiente para a multiplicação de matrizes suficientemente grandes, de forma que para matrizes pequenas é mais eficiente trocá-lo pelo método tradicional.

2.5 Considerações sobre o Capítulo

Neste capítulo foram apresentados os conceitos relacionados ao paralelismo de tarefas e ao escalonamento por roubo de tarefas. Com relação ao paralelismo de tarefas foram discutidos aspectos como a granularidade das tarefas e o escalonamento que impactam diretamente no desempenho e nos desafios envolvidos na implementação de sistemas com suporte ao paralelismo de tarefas. Também foram apresentadas algumas ferramentas com suporte ao paralelismo de tarefas como Cilk, Intel TBB, OpenMP, XKaapi e StarPU.

Cilk é uma das primeiras ferramentas a implementar o paradigma que é conhecido atualmente como paralelismo de tarefas, fazendo uso do algoritmo de escalonamento por roubo de tarefas (*work stealing*). OpenMP é a ferramenta mais utilizada para programação paralela em sistemas de memória compartilhada. Até a versão 2.5 ofereceu suporte somente ao paralelismo de laços, porém, a partir da versão 3.0 oferece suporte ao paralelismo de tarefas. Intel TBB é uma biblioteca de templates que permite utilização tanto de paralelismo de laços quanto de tarefas. Essas três ferramentas são voltadas somente para sistemas de memória compartilhada com CPUs *multicore*. Cilk e Intel TBB possuem em comum a característica de utilizarem o algoritmo de escalonamento por roubo de tarefas enquanto as implementações de OpenMP fazem uso de técnicas mais simples como *round-robin*, uma vez que a política de escalonamento não é definida pela especificação OpenMP.

As ferramentas acadêmicas XKaapi e StarPU têm por objetivo oferecer suporte a sistemas com arquiteturas híbridas, que fazem uso tanto de CPUs *multicore* como de GPUs. XKaapi implementa o escalonamento por roubo de tarefas, porém na versão atual os roubos só são realizados entre recursos do mesmo tipo (p. ex. somente entre GPUs). StarPU oferece um conjunto de políticas de escalonamento, entretanto a de melhor desempenho (HEFT) é baseada em modelos para previsão de tempo de execução e de transferência de dados. Como característica comum, as duas ferramentas permitem que uma tarefa possua múltiplas implementações, cada uma delas específica para um determinado recurso de processamento.

Todas as ferramentas apresentadas, com exceção do StarPU, permitem que uma tarefa em execução crie tarefas filhas dinamicamente. Essas tarefas filhas não são necessariamente processadas no mesmo recurso de processamento que executa a tarefa pai. Entretanto, nenhuma das ferramentas discutidas apresenta simultaneamente suporte ao paralelismo de tarefas com criação dinâmica de tarefas, escalonamento por roubo de tarefas e suporte a execução de tarefas em CPUs e GPUs.

O Capítulo 3, a seguir, apresenta uma proposta para execução de tarefas baseada nos conceitos de paralelismo de tarefas e escalonamento por *work stealing* aplicados ao cenário de arquiteturas computacionais compostas por recursos híbridos como CPUs *multicores* e GPUs.

3 WORMS: ESCALONAMENTO POR ROUBO DE TAREFAS EM SISTEMAS MULTI-CPU E MULTI-GPU

O Capítulo 2 apresentou os conceitos de paralelismo de tarefas e de escalonamento por roubo de tarefas. Além disso, na Seção 2.3 foram apresentadas ferramentas que oferecem suporte ao paralelismo de tarefas. Conforme discutido anteriormente nenhuma dessas ferramentas reúne as características de suporte a paralelismo de tarefas, escalonamento por roubo de tarefas e suporte a recursos CPU e GPU simultaneamente.

Este capítulo apresenta o *middleware* WORMS (WORk stealing scheduling for Multi-CPU/GPU Systems), que implementa o suporte ao paralelismo de tarefas em arquiteturas híbridas juntamente com o escalonamento por meio de um algoritmo de roubo de tarefas.

O WORMS possui como características:

- permitir múltiplas implementações para cada tarefa para execução em CPUs e GPUs;
- permitir que uma tarefa em execução crie tarefas filhas dinamicamente;
- possibilitar a computação de tarefas com dependências totalmente estritas (*fully strict*[1]) em CPUs *multicore* e GPUs simultaneamente;
- escalonar as tarefas entre as CPUs e as GPUs por meio da técnica de roubo de tarefas.

O modelo proposto para o *middleware* WORMS é apresentado na Seção 3.1. Foram implementadas duas versões da abordagem proposta para o *middleware*. A primeira versão, apresentada na Seção 3.2.1, é caracterizada pela utilização de duas filas de tarefas. A versão atual, que utiliza apenas uma fila de tarefas, é apresentada na Seção 3.2.2. Essa versão apresenta otimizações em algumas limitações identificadas na versão inicial. A implementação do *middleware* é abordada na Seção 3.3. Já a Seção 3.4 apresenta a API em nível de usuário para programação utilizando o WORMS. Por fim, a Seção 3.5 discute algumas das limitações técnicas identificadas na versão atual do WORMS.

3.1 Modelo

O modelo proposto para o WORMS possui três partes principais: o componente *Manager*, o componente *Task* e um conjunto de componentes *TaskProcessor*.

No WORMS, assim como em outras ferramentas apresentadas no capítulo anterior que implementam escalonamento por roubo de tarefas, cada recurso de processamento possui

[1] Tarefas com dependências totalmente estritas (*fully strict*) são aquelas em que a única relação de dependência possível de uma tarefa é com suas tarefas filhas (BLUMOFE; LEISERSON, 1999).

uma lista de tarefas prontas para execução. As unidades de execução do WORMS são chamadas de *TaskProcessors*. Além disso, cada tarefa WORMS pode criar novas tarefas filhas, bem como esperar pela conclusão das tarefas filhas criadas.

O *Manager* é o componente responsável pela criação dos componentes *TaskProcessor*, submissão de novas *Tasks* e pela sincronização entre as *Tasks* pai e as *Tasks* filhas.

Uma *Task* é o componente que representa uma unidade de computação. Cada *Task* possui, pelo menos, três métodos virtuais: um método *runCPU()* que contém a implementação para execução em CPU, um método *runGPU()* que contém a implementação para execução em GPU e um método *runTermination()* para a terminação da tarefa. Apenas a implementação do método *runCPU()* é obrigatória, a implementação dos métodos *runGPU()* e *runTermination()* é opcional. O método de terminação de uma tarefa permite a execução de alguma computação após o ponto de sincronização entre a tarefa pai e as tarefas filhas, utilizando, por exemplo, os resultados calculados na execução dessas tarefas filhas.

Os *TaskProcessors* são os componentes utilizados para executar as *Tasks*. Cada *TaskProcessor* possui uma fila de duas pontas, ou deque (*double-ended queue*), de *Tasks* prontas para serem executadas. A deque de *Tasks* para execução de um *TaskProcessor* pode ser acessada por outros *TaskProcessors* que estejam tentando roubar tarefas. Os acessos de um *TaskProcessor* a sua própria deque tanto para inserção quanto para a remoção de elementos são feitos sempre na extremidade inicial da deque. Já os acessos feitos por outro *TaskProcessor* em caso de tentativa de roubo são realizados sempre na extremidade final da deque.

Na primeira versão do *middleware*, além dessa deque os *TaskProcessors* também possuem uma fila privada para *Tasks* em espera. Cada componente *TaskProcessor* é associado a um núcleo de CPU existente na arquitetura. Se existirem GPUs disponíveis, alguns *TaskProcessors* são associados simultaneamente a um núcleo CPU e a uma das GPUs existentes.

3.2 Funcionamento

Esta Seção descreve as características de funcionamento comuns às duas versões do WORMS. A subseção 3.2.1 apresenta os aspectos de funcionamento específicos da primeira versão do *middleware*, enquanto a subseção 3.2.2 aborda as especificidades da versão atual do WORMS.

No início da execução do *middleware* WORMS o componente *Manager* cria uma *thread* para cada núcleo/CPU existente na arquitetura em questão. Para cada uma dessas *threads* é criado e associado um *TaskProcessor*. Quando existe uma GPU disponível no sistema um dos *TaskProcessors* é associado a essa GPU. Caso existam mais GPUs na arquitetura em questão outros *TaskProcessors* são designados para comandá-las.

Dessa forma, os *TaskProcessors* que gerenciam simultaneamente um dos núcleos da CPU e uma GPU possuem capacidade de processamento diferente dos demais que estão associados apenas aos *cores* da CPU. Quando um *TaskProcessor* associado a uma GPU for executar uma tarefa que não tenha o método *runGPU()* implementado, a execução é feita em CPU, ou seja, um *TaskProcessor* associado a uma GPU também pode executar tarefas em CPU quando necessário. Na implementação do método *runGPU()* devem ser feitas as transferências de memória necessárias para que a execução da tarefa possa ser feita na GPU e para que os dados sejam recuperados após o fim dessa execução.

3.2.1 Versão com duas filas

Esta subseção descreve o funcionamento da primeira versão do WORMS. Essa versão (PINTO; MAILLARD, 2012a) é caracterizada pela presença de duas filas de tarefas por *TaskProcessor*: uma deque para tarefas prontas para execução e uma fila simples para tarefas em espera.

Durante sua execução, cada componente *Task* pode criar e submeter novas *Tasks* filhas para serem executadas, além disso, a *Task* pai pode executar uma chamada de espera assíncrona. Essa chamada permite que essa *Task* seja colocada em uma fila de espera. A partir do momento em que todas as *Tasks* filhas submetidas tenham terminado sua execução, a *Task* pai pode ser retirada da fila de espera e então, após a retirada dessa fila, o método de terminação da *Task* pai pode ser executado. Como a fila de espera de cada *TaskProcessor* é privada, o método de terminação de uma *Task* só pode ser executado pelo *TaskProcessor* que iniciou a execução da *Task*. Dessa forma, os estados possíveis para uma *Task* nesta versão do WORMS são:

- *Task* pronta para execução: a *Task* foi criada e está na deque de algum *TaskProcessor* aguardando para ser executada;
- *Task* em execução: a *Task* que foi retirada da deque de algum *TaskProcessor* e está executando o seu método de processamento ou de terminação;
- *Task* em espera: a *Task* criou *Tasks* filhas, foi colocada na fila de *Tasks* em espera e está aguardando pelo fim das *Tasks* filhas para que possa ser concluída ou ter sua terminação executada;
- *Task* concluída: a *Task* foi marcada como concluída após ter obrigatoriamente executado algum método de processamento (*runCPU()* ou *runGPU()*), todas as eventuais *Tasks* filhas criadas terem sido concluídas e ter executado, se existir, o seu método de terminação (*runTermination()*).

O fluxograma da Figura 3.1 ilustra de forma simplificada o comportamento do algoritmo executado por um *TaskProcessor* para a processar uma *Task*. O método de processamento referenciado nesse fluxograma pode ser tanto o método *runCPU()* quanto o método *runGPU()* no caso de o *TaskProcessor* também estar associado a uma GPU.

A Figura 3.2 apresenta o fluxograma que mostra de forma simplificada o comportamento do algoritmo para buscar uma nova tarefa para execução. Nesse fluxograma é descrita a ordem de busca: primeiramente é verificada a deque do próprio *TaskProcessor*, após é feita uma tentativa de roubo da deque de outro *TaskProcessor* e por último é verificada a fila de *Tasks* em espera. O *TaskProcessor* vítima da tentativa de roubo é escolhido por meio de estratégia aleatória.

3.2.2 Versão com uma fila

Esta subseção aborda o funcionamento da versão atual do *middleware* WORMS. Essa versão caracteriza-se pela necessidade de apenas uma fila de tarefas por *TaskProcessor*.

A principal modificação introduzida é a mudança no algoritmo de gerenciamento das *Tasks* que estão esperando pela execução de *Tasks* filhas. Essa mudança possibilita a eliminação da fila de *Tasks* em espera. Dessa forma, cada *TaskProcessor* possui apenas uma deque de tarefas prontas para execução. Essas modificações visaram melhorar o desempenho do *middleware* em aplicações que geram um grande número de tarefas.

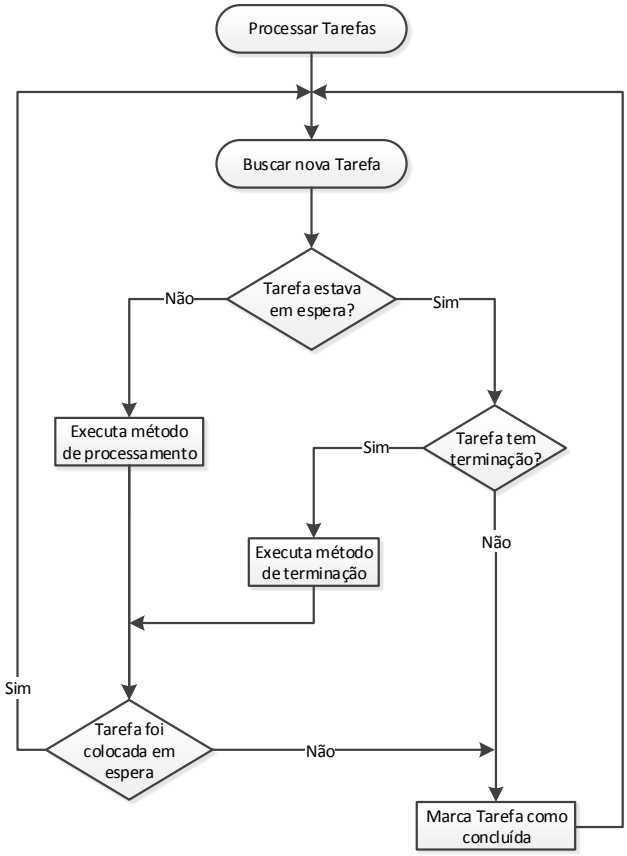


Figura 3.1: Comportamento de um *TaskProcessor* para executar tarefas.

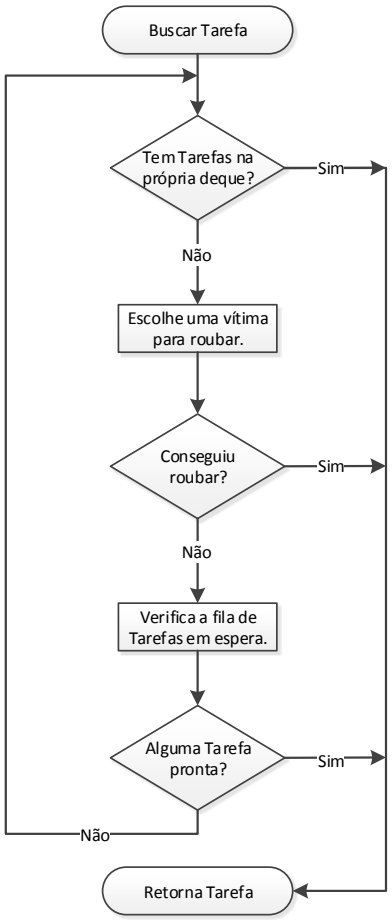


Figura 3.2: Comportamento de um *TaskProcessor* para buscar tarefas.

Os procedimentos executados pelo componente *Manager* na inicialização do *middleware* no contexto da criação das *threads*, dos *TaskProcessors*, e da associação desses *TaskProcessors* aos núcleos da CPU e às GPUs não foi modificado em relação a primeira versão do WORMS. Também foi mantida a capacidade de um *TaskProcessor* associado a uma GPU executar em CPU as tarefas que não tenham o método *runGPU()* implementado.

Assim como na versão anterior, cada componente *Task* pode criar e submeter novas *Tasks* filhas para serem executadas bem como executar uma chamada de espera assíncrona. Porém, essa chamada de espera não implica na colocação da *Task* pai em uma fila de espera mas sim no incremento de uma variável contadora associada a essa *Task*.

A variável contadora associada a uma *Task* tem por objetivo indicar a atividade da *Task*, ou seja, se a *Task* tem trabalho a ser executado. Esse trabalho a ser executado pode ser tanto os métodos da própria *Task* (processamento e terminação) quanto os métodos das *Task* filhas eventualmente criadas. O contador de atividade de uma *Task* é modificado nas seguintes situações:

- Inicializado em 1 quando a *Task* é criada;
- Incrementado quando a *Task* submete uma *Task* filha para execução;
- Incrementado quando o método de terminação da *Task* vai ser executado;
- Decrementado quando termina a execução do método de processamento da *Task*;
- Decrementado quando termina a execução do método de terminação da *Task*;
- Decrementado quando uma *Task* filha é marcada como concluída.

O método de terminação de uma *Task*, se existir, somente pode ser executado após o contador da atividade da *Task* ser zerado. Essa condição é atendida quando termina a execução do método de processamento da *Task* e quando todas as possíveis *Tasks* filhas criadas tenham sido concluídas. Dessa forma, o método de terminação de uma *Task* não é necessariamente executado no mesmo *TaskProcessor* que iniciou a execução da *Task*. Esse método de terminação de uma *Task* pai pode ser executado pelo *TaskProcessor* que executou a última *Task* filha desta *Task* pai.

Os estados possíveis de uma *Task* na versão do WORMS com uma fila são:

- *Task* pronta para execução: uma *Task* que foi criada e está na deque de algum *TaskProcessor* esperando para ser executada;
- *Task* ativa: uma *Task* que possui sua variável contadora de atividade com valor diferente de zero e não está na deque de nenhum *TaskProcessor*. Essa *Task* pode estar executando seu método de processamento, executando seu método de terminação e/ou esperando que todas as *Tasks* filhas sejam concluídas;
- *Task* concluída: uma *Task* que foi marcada como concluída após ter executado algum dos seus métodos de processamento (*runCPU()* ou *runGPU()*), serem concluídas todas as *Tasks* filhas criadas durante sua execução e ter executado seu método de terminação (*runTermination()*), se esse existir.

O fluxograma da Figura 3.3 apresenta o comportamento do algoritmo de um *TaskProcessor* para processar uma *Task*. O procedimento de busca de nova tarefa, ilustrado na Figura 3.4, foi simplificado em relação a versão inicial do WORMS através da eliminação da fila de tarefas em espera. Assim como na versão inicial, a escolha do *TaskProcessor* vítima da tentativa de roubo é feita de forma aleatória, visando minimizar o custo de escalonamento.

As demais modificações relevantes realizadas no algoritmo nessa versão em relação a versão com duas filas são: a manipulação da variável contadora de atividade (operações de verificação, incremento e decremento) e a verificação que permite identificar se a *Task* em questão foi a última filha a ser concluída. Sempre que uma *Task* for concluída é realizada uma verificação para saber se essa *Task* foi a última *Task* filha a ser concluída. Em caso afirmativo, o método de terminação da *Task* pai é executado. Esse comportamento é recursivo, ou seja, se a *Task* pai for a última *Task* filha de outra *Task* pai a ser executada, o método de terminação da *Task* avô também deve ser executado. O comportamento para execução do método de terminação da *Task* pai é apresentado no fluxograma da Figura 3.5

3.3 Implementação

A abordagem WORMS apresentada neste trabalho implementa a execução de tarefas paralelas utilizando o escalonamento por meio de roubo de tarefas conforme descrito nas seções anteriores. Essa implementação é feita em linguagem C++, seguindo o paradigma de programação orientado a objetos. Os componentes descritos anteriormente são implementados como classes C++, sendo as classes que representam o componente *Task* implementadas como classes abstratas. A biblioteca Boost C++[2] (LING, 2011; DUFFY, 2010) é utilizada para gerenciamento das *threads* e controle dos pontos de bloqueio e sincronização. O ambiente CUDA[3] (*Compute Unified Device Architecture*) é utilizado para a programação das tarefas que executam em GPUs (NVIDIA, 2011).

3.4 API

A API em nível de usuário para a programação utilizando o WORMS é inspirada nos conceitos de outras ferramentas que implementam paralelismo de tarefas como aquelas apresentadas no Capítulo 2. Usualmente, existe um objeto da classe *Manager* por programa. Esse objeto é utilizado para recuperar as referências aos objetos *TaskProcessor* e para submeter e gerenciar os objetos *Task*. Quando o objeto *Manager* é inicializado, são criados os *TaskProcessors* e as *threads* associadas a esses, e é submetida a tarefa principal (*MainTask*). Essa tarefa passa a ser executada por um dos *TaskProcessors* e é a partir

[2]Boost C++ é um conjunto de bibliotecas portáveis que estendem funcionalidades da linguagem C++. As bibliotecas Boost oferecem suporte a diversos recursos como: programação concorrente, manipulação de tempo, de strings, de sistemas de arquivos, de imagens e de expressões regulares (BOOST, 2012). Neste trabalho, foi adotada a biblioteca *Boost Thread* para suporte à programação concorrente (WILLIAMS, 2012).

[3]O ambiente CUDA é uma plataforma para programação paralela de propósito geral em GPUs fabricadas pela NVIDIA. A programação em CUDA é baseada em uma extensão da linguagem C, que permite codificar um *kernel* que é uma função que será compilada para execução na GPU. A API base do CUDA também disponibiliza funções para alocação e cópias de memória entre a GPU e o sistema que a comanda. O ambiente CUDA inclui ainda um conjunto de bibliotecas otimizadas como Thrust, CUFFT e CUBLAS (NVIDIA Developer Zone, 2012a).

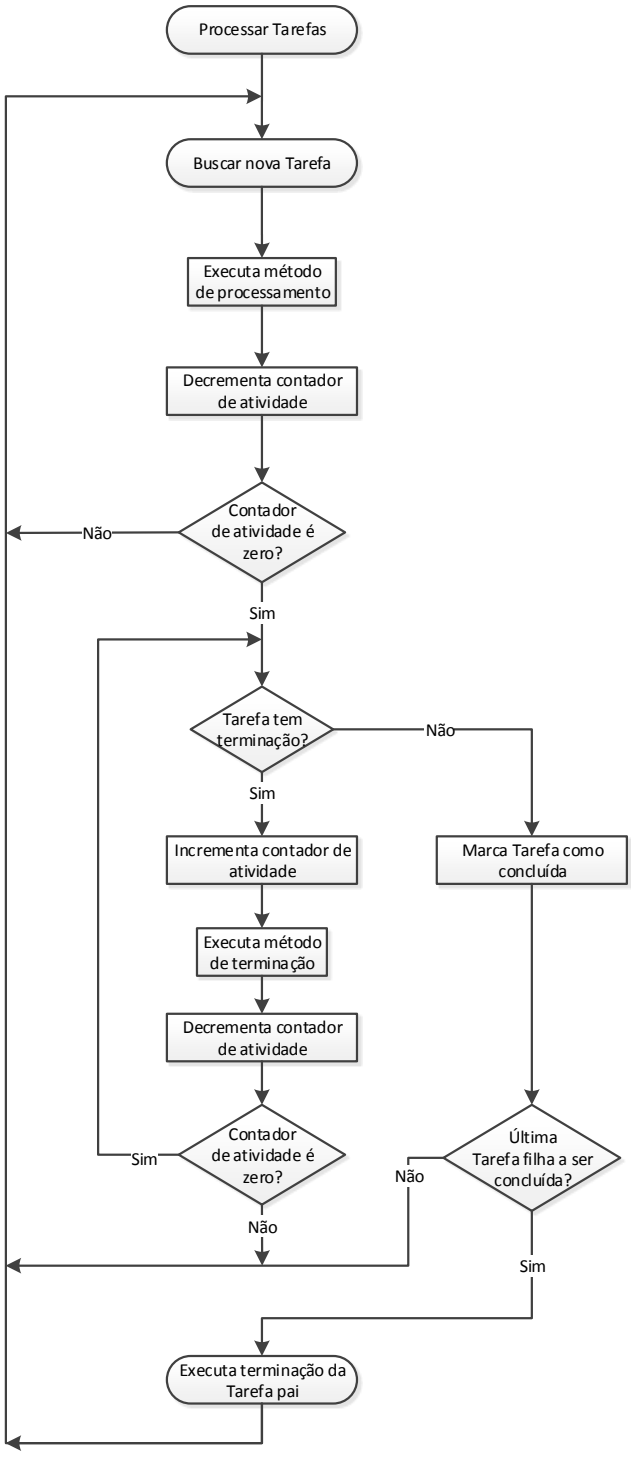


Figura 3.3: Comportamento de um *TaskProcessor* para processar tarefas na versão atual do WORMS.

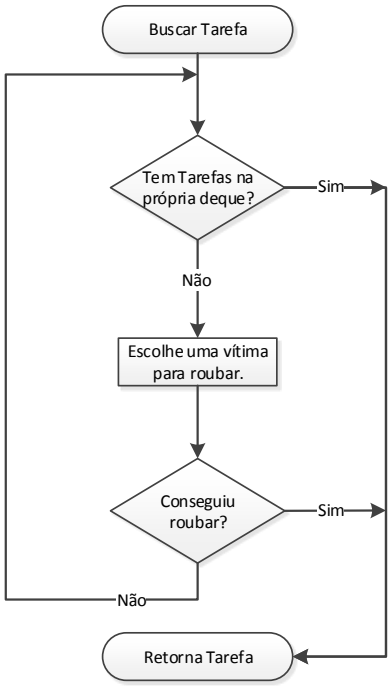


Figura 3.4: Comportamento de um *TaskProcessor* para buscar tarefas na versão atual do WORMS.

dela que são criadas as demais tarefas. A execução do programa termina quando a tarefa principal é marcada como concluída.

A API encapsula as chamadas ao *Manager* e aos *TaskProcessors*. Dessa forma, o usuário interage apenas com as três classes relacionadas ao componente *Task*:

- *TaskCPU:* Classe abstrata que possui um método virtual puro utilizado para processamento na CPU (*runCPU()*) e um método virtual de terminação (*runTermination()*);
- *TaskCPUGPU:* Subclasse derivada de *TaskCPU* que adiciona o método virtual puro para processamento em GPU (*runGPU()*);
- *MainTask:* Subclasse derivada de *TaskCPU* utilizada para iniciar a computação utilizando o WORMS.

A programação de uma aplicação em nível de usuário sempre é feita a partir da *MainTask* que deve ser utilizada como equivalente a uma função *main* de um programa C/C++ tradicional. A classe *MainTask* é uma subclasse da classe *TaskCPU* que têm acesso aos parâmetros passados pela linha de comando.

As tarefas específicas criadas para uma aplicação devem ser compostas de uma classe que deve estender a classe abstrata *TaskCPU* ou uma das subclasses *TaskCPUGPU* ou *MainTask*. Além disso, deve ser implementado obrigatoriamente o método *runCPU()* e opcionalmente os métodos *runGPU()* e *runTermination()*.

A Tabela 3.1 apresenta a descrição dos métodos da API do WORMS que possibilitam a manipulação de *Tasks* paralelas. Esses métodos devem ser invocados a partir dos métodos de processamento ou de terminação da *Task*.

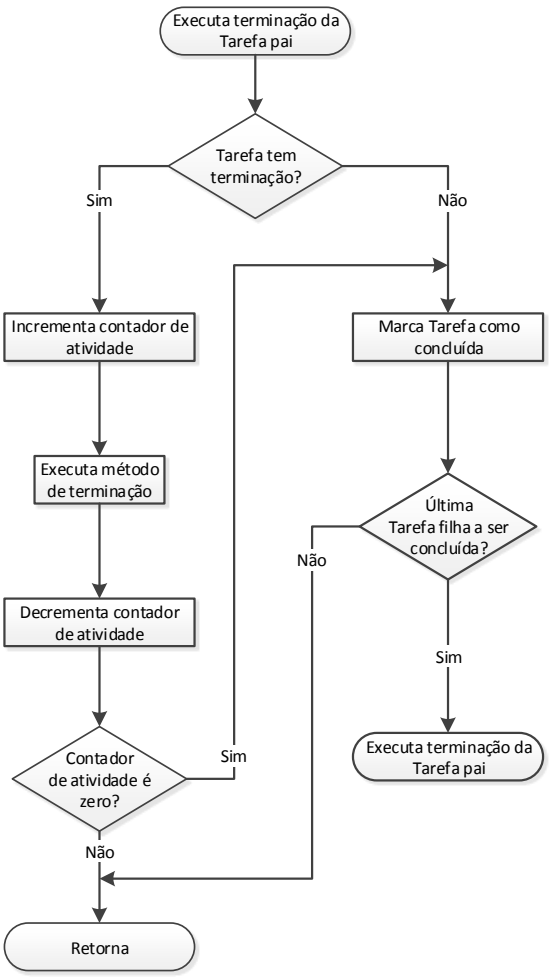


Figura 3.5: Comportamento de um *TaskProcessor* para executar o método de terminação de uma tarefa pai.

Método	Descrição
taskSub(TaskCPU t)*	submete a *Task* t para a deque do *TaskProcessor* que está executando a *Task* que faz a chamada ao método;
taskSubGeneral(TaskCPU t)*	submete a *Task* t para a deque de algum dos *TaskProcessors*;
asyncWaitChildrenTasks()	indica que a *Task* que chama o método deve aguardar até o fim do processamento das *Tasks* filhas para ser concluída;
asyncWaitChildrenTasksRunTerm()	indica que a *Task* que chama o método deve aguardar até o fim do processamento das *Tasks* filhas para então ter seu método de terminação executado.

Tabela 3.1: Métodos da API do WORMS para manipulação de tarefas.

Os métodos virtuais que devem ser implementados pelo usuário da API são apresentados na Tabela 3.2. A implementação do método *runGPU()* para as *Tasks* derivadas da classe *TaskCPUGPU* deve incluir as transferências de dados necessárias para a execução na GPU e a recuperação desses dados após a computação.

Método virtual	Descrição	Implementação
runCPU()	método de processamento da *Task* para ser executado em CPU.	Obrigatória para todas as *Tasks*
runGPU()	método de processamento da *Task* para ser executado em GPU.	Obrigatória para *Tasks* derivadas de *TaskCPUGPU*
runTermination()	método de terminação da *Task*.	Opcional

Tabela 3.2: Métodos virtuais da API do WORMS.

Exemplos de uso da API do WORMS e dos métodos descritos nas Tabelas 3.1 e 3.2 são apresentados nos Códigos 3.1 e 3.2. O Código 3.1 ilustra um exemplo de definição de tarefa no WORMS. Nesse exemplo, a classe *Fib* é uma subclasse de *TaskCPU*, logo essa tarefa será executada somente em CPU.

Código 3.1: Exemplo de definição de uma tarefa com WORMS.

```
class Fib : public TaskCPU {
public:
   Fib(int* res, int n);
   virtual void runCPU();
   virtual void runTermination();
private:
   int rx, ry, n;
   int* result;
};
```

O exemplo apresentado no Código 3.2 mostra a implementação do métodos de processamento (*runCPU()*) e de terminação (*runTermination()*) da classe *Fib* cuja definição é apresentada no Código 3.1. As chamadas aos métodos *taskSub* e *asyncWaitChildrenTasksRunTerm* são assíncronas. A chamada ao método *asyncWaitChildrenTasksRunTerm*

torna explicito que a *Task* deverá executar algum processamento após a conclusão da execução das *Tasks* filhas criadas, que irão calcular os valores *rx* e *ry*. O processamento a ser executado após o termino das *Tasks* filhas, a soma entre *rx* e *ry*, é implementado no método *runTermination*.

Código 3.2: Exemplo de definição de uma tarefa com WORMS.

```
void Fib::runCPU() {
    if (n < 2) {
        *result = n;
    } else {
        Fib *fb;
        fb = new Fib(&rx, n - 1);
        this->taskSub(fb);
        fb = new Fib(&ry, n - 2);
        this->taskSub(fb);
        this->asyncWaitChildrenTasksRunTerm();
    }
}

void Fib::runTermination() {
    *result = rx + ry;
}
```

3.5 Desafios de implementação e limitações técnicas identificadas

A implementação do *middleware* descrito neste Capítulo permitiu a identificação de diversos desafios e limitações solucionados em parte ou contornados. Apesar da API de nível de usuário ter sido inspirada na API da ferramenta Cilk, em especial nos métodos de criação/submissão de tarefas filhas e de sincronização da tarefa pai com as tarefas filhas, a implementação do ambiente de execução do *middleware* WORMS não seguiu todas as decisões técnicas conhecidas sobre a implementação do Cilk.

As decisões adotadas na implementação do WORMS decorrem, em parte, de aspectos básicos de projeto como a escolha da linguagem de programação C++ e o fato de o *middleware* ser uma biblioteca, ao contrário do Cilk que é escrito em C e é implementado como um compilador.

Outro ponto diferencial em relação ao Cilk na implementação do ambiente de execução é a ordem de execução das tarefas filhas. No WORMS as tarefas filhas criadas são empilhadas na deque e a tarefa pai continua sua execução, em um conceito referenciado na literatura como *help-first* (GUO et al., 2009). De outra forma, Cilk utiliza o conceito *work-first* na qual a tarefa filha passa a ser executada e a tarefa pai é empilhada. Maiores detalhes sobre os conceitos de *help-first* e *work-first* e suas implicações são apresentados em (GUO et al., 2009).

Por seguir o paradigma de programação orientado a objetos, no WORMS a estrutura utilizada para representar uma tarefa impacta em um consumo de memória maior, o que pode provocar sobrecustos em aplicações que criem um número elevado de tarefas. Essa limitação foi minimizada na atual versão do *middleware* devido à eliminação da fila de tarefas em espera nos *TaskProcessors* e da lista de tarefas filhas. O experimento A.3.1 do Apêndice A apresenta uma comparação entre o desempenho das duas versões do WORMS ao gerenciar um número elevado de tarefas.

Além do tamanho da estrutura de representação da tarefa, outros aspectos que podem ocasionar sobrecusto no gerenciamento de um número de tarefas elevado são a adoção

do conceito de *help-first*, conforme mostrado em (GUO et al., 2009) e a sincronização no acesso às deques.

Nas duas versões do *middleware* WORMS o acesso em exclusão mútua às deques de cada recurso de processamento é realizado por meio de primitivas bloqueantes do tipo *lock*. Dessa forma, antes de qualquer acesso à uma deque esta é totalmente bloqueada, mesmo que o acesso seja a extremidades diferentes da deque. A solução adotada por Cilk para reduzir os bloqueios nos acessos mutualmente exclusivos às deques é adoção do protocolo de exclusão mútua THE (FRIGO; LEISERSON; RANDALL, 1998) baseado na proposta apresentada por Dijkstra em (DIJKSTRA, 1965). Uma vez que as implementações do *middleware* WORMS utilizam o *container* deque disponibilizado pela biblioteca Boost C++, uma solução viável em futuro próximo é a utilização do pacote de estruturas de dados *Lockfree* (BLECHMANN, 2013) incluído na versão 1.53.0 do Boost C++. Esse pacote ainda não suporta *containers* do tipo deque, porém, a inclusão deste suporte está prevista para a próxima versão da biblioteca (BLECHMANN, 2013).

3.6 Conclusões sobre o Capítulo

O WORMS é uma proposta que implementa o suporte ao paralelismo de tarefas em arquiteturas híbridas escalonando as tarefas paralelas por meio de um algoritmo de roubo de tarefas (*work stealing*). Foram apresentados o modelo proposto para o *middleware*, as duas versões implementadas, as características da implementação, a API em nível de usuário e alguns desafios de implementação e limitações técnicas identificadas.

As mudanças introduzidas na versão atual do WORMS em relação a versão anterior possibilitaram a melhora do desempenho na execução de aplicações que criam um número elevado de tarefas. Isso foi possível através de otimizações no consumo de memória dos *TaskProcessors* e das estruturas que representam as tarefas.

No Capítulo 4 é apresentada uma avaliação experimental do desempenho da proposta implementada pelo WORMS. Nessa avaliação o desempenho obtido pelo WORMS é comparado ao desempenho apresentado por ferramentas de referência para processamento em CPU e GPU.

4 AVALIAÇÃO EXPERIMENTAL

O Capítulo 3 apresentou o *middleware* WORMS que oferece suporte ao paralelismo de tarefas em ambientes híbridos CPU e GPU. Este capítulo apresenta uma avaliação experimental da abordagem descrita no Capítulo 3. A Seção 4.1 apresenta as plataformas computacionais utilizadas para execução dos testes apresentados neste Capítulo. A Seção 4.3 aborda a avaliação de desempenho do ambiente WORMS utilizando um conjunto de aplicações de teste em três cenários de execução: somente com a CPU *multicore*, com CPU *multicore* e GPU e somente com GPU. As Seções 4.4 e 4.5 apresentam uma avaliação do desempenho de ferramentas de referência como Cilk para execuções em CPUs *multicore* e CUDA para execuções somente em GPUs utilizando as mesmas aplicações testadas na Seção 4.3. Por fim, a Seção 4.6 apresenta uma comparação entre os resultados do WORMS mostrados na Seção 4.3 com os resultados das ferramentas de referência apresentados nas Seções 4.4 e 4.5.

4.1 Plataformas de Teste

Para execução dos testes de avaliação de desempenho foram utilizados dois ambientes computacionais: o nó de processamento *Bugio* do Grupo de Processamento Paralelo e Distribuído (GPPD) e um dos nós do *cluster Newton* do Centro Nacional de Supercomputação (CESUP) da UFRGS. O CESUP/UFRGS é uma unidade do Sistema Nacional de Processamento de Alto Desempenho (SINAPAD).

A plataforma *Bugio* é composta por um processador Intel Core i7 930 quad-core com 2.80GHz de frequência de *clock* e 12GB de memória RAM e por uma GPU NVIDIA GTX480 com 480 CUDA *cores* com 1.4GHz de frequência de *clock* e 1.5GB de memória RAM. Essa GPU é conectada ao sistema através de barramento PCI-E 2.0. Esse ambiente utiliza o sistema operacional Ubuntu 11.10 GNU/Linux, com *kernel* Linux 3.0.0, driver NVIDIA 4.1, CUDA 4.1, compilador gcc versão 4.4.6 e biblioteca Boost C++ versão 1.49. A biblioteca ATLAS 3.9.63 (*Automatically Tuned Linear Algebra Software*) (ATLAS, 2012; WHALEY; DONGARRA, 1998) é utilizada para oferecer suporte a sub-rotinas BLAS. A Figura 4.1 ilustra a topologia hierárquica da arquitetura da plataforma *Bugio*, detalhando a configuração de memória *cache* entre os núcleos da CPU e as GPUs conectadas ao barramento. Essa topologia foi gerada pelo *software hwloc* (Open MPI, 2013).

A plataforma *Newton* é composta por dois processadores AMD Opteron 2427 six-core[1] com 2.20GHz de frequência de *clock* e 16GB de memória RAM e por duas GPUs

[1] Apesar de a plataforma *Newton* contar com 12 núcleos de CPU, os testes foram executados com até 10 núcleos em virtude da demanda de usuários do *cluster* e das opções permitidas pelo sistema de submissão de *jobs* que não possibilitaram a utilização simultânea dos 12 núcleos.

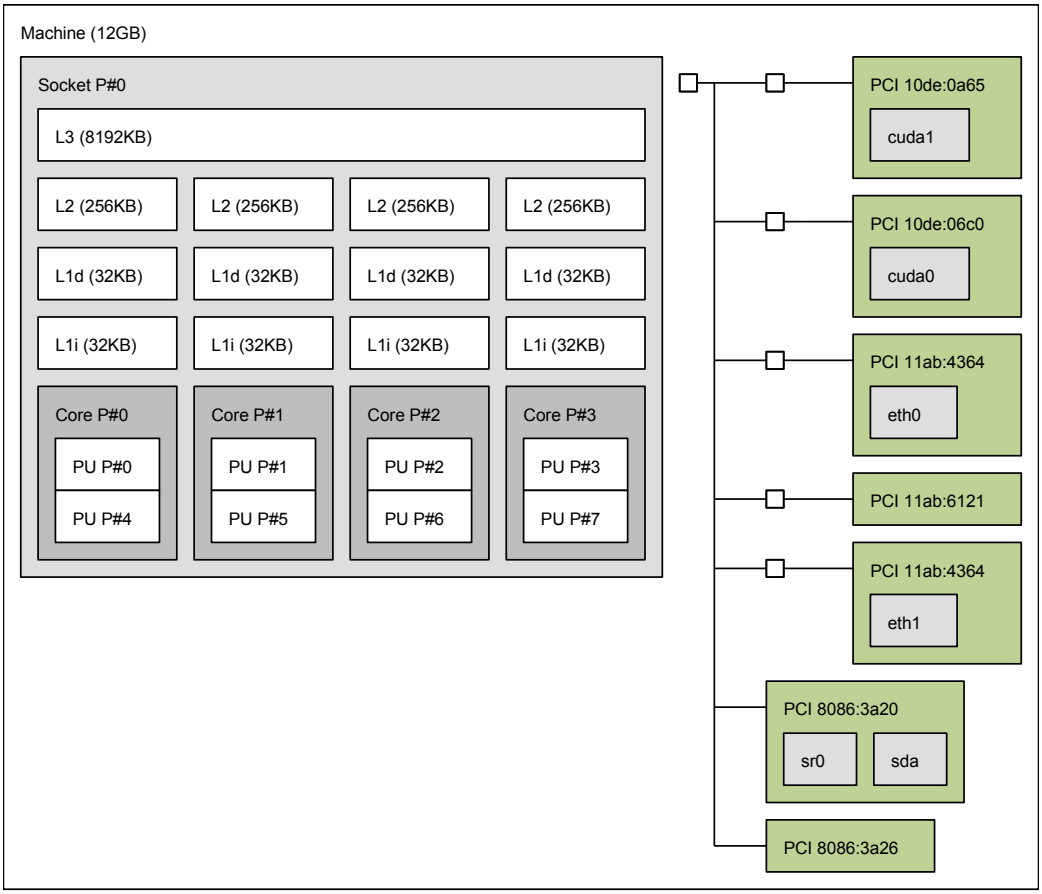


Figura 4.1: Topologia hierárquica da plataforma Bugio. O dispositivo referenciado como *cuda0* é a GPU GTX480 dedicada para processamento. O dispositivo referenciado como *cuda1* é a GPU utilizada somente para vídeo.

NVIDIA Tesla S1070 com 240 CUDA *cores* cada uma, com frequência de *clock* de 1.296 a 1.44GHz e 4GB de memória RAM. As GPUs são conectadas ao sistema através de barramento PCI-E 2.0. Essa plataforma utiliza o sistema operacional Red Hat Enterprise Linux 5.5, com *kernel* Linux 2.6.18, driver NVIDIA 3.2, CUDA 3.2, compilador gcc versão 4.1.2 e biblioteca Boost C++ versão 1.52. A biblioteca de sub-rotinas BLAS disponível é a ACML (*AMD Core Math Library*) (AMD, 2012). A Figura 4.2 ilustra a topologia hierárquica da arquitetura da plataforma *Newton*, detalhando a configuração de memória *cache* entre os núcleos das CPUs e as GPUs conectadas ao barramento. Essa topologia foi gerada pelo *software hwloc* (Open MPI, 2013).

Nos experimentos apresentados neste Capítulo, a plataforma *Bugio* é utilizada como ambiente multi-CPU/mono-GPU[2]. A plataforma *Newton* é utilizada como ambiente para execução de testes multi-CPU/multi-GPU.

[2]Neste trabalho, consideramos como multi-CPU qualquer teste que utilize mais de um núcleo de uma CPU *multicore*.

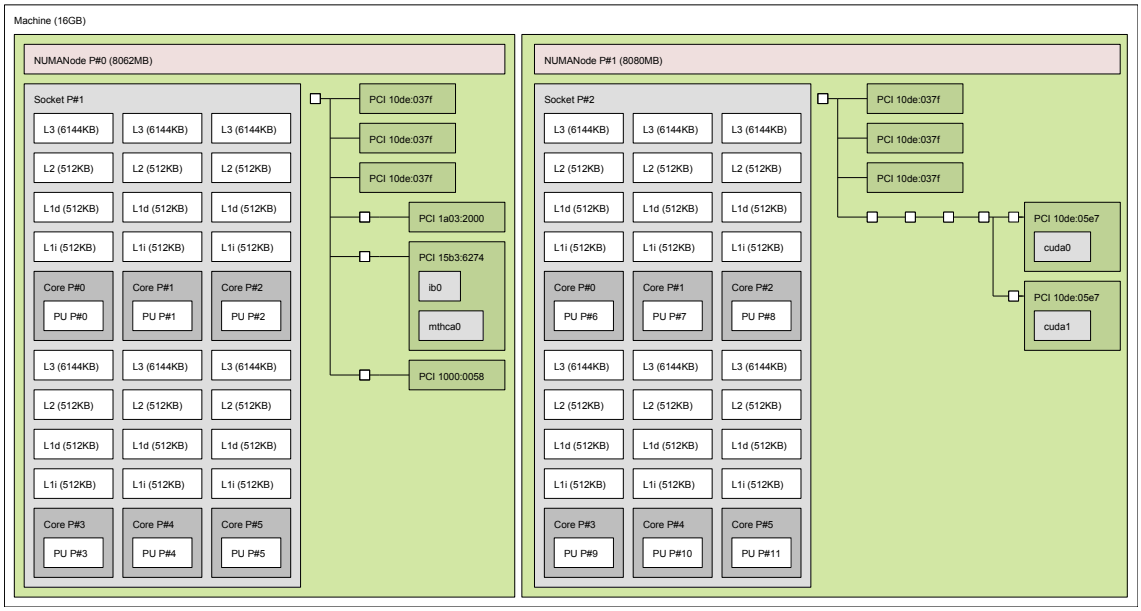


Figura 4.2: Topologia hierárquica da plataforma Newton. Os dispositivos referenciados como *cuda0* e *cuda1* representam as GPUs dedicadas para processamento.

4.2 Protocolo Experimental

Foram utilizadas três aplicações de teste: transformação, ordenação e multiplicação de matrizes. Os três algoritmos apresentam características distintas de computação. A aplicação Transformação possui a parte mais intensa da sua computação realizada nas tarefas folhas, após a conclusão do processamento das tarefas folhas não são necessários novos cálculos. A aplicação Ordenação apresenta comportamento distinto pois após conclusão da computação das tarefas folhas os resultados ainda devem ser combinados para que a computação seja concluída, além disso, a aplicação acessa áreas descontínuas de memória. Já a aplicação Multiplicação de Matrizes apresenta a maior carga computacional entre as três aplicações. Além de gerar um número maior de tarefas folhas, possui computação intensiva tanto durante a execução das tarefas folhas quanto no processamento dos resultados parciais computados por essas folhas.

Cada experimento foi executado com no mínimo 50 repetições. Os resultados ilustrados nos gráficos representam a média dessas 50 execuções juntamente com o desvio padrão. Todos os testes foram compilados utilizando o conjunto de otimizações `O3` do compilador `GCC`.

4.3 Avaliação do *middleware* WORMS

Na avaliação dos experimentos com o *middleware* WORMS são utilizadas duas configurações: uma somente com CPUs e outra reunindo CPUs e GPUs. Os experimentos apresentados nesta seção têm por objetivo validar o funcionamento do *middleware* e o impacto a inclusão de GPUs no desempenho das aplicações.

4.3.1 Multiplicação de Matrizes por Algoritmo de Strassen

A aplicação de teste utilizada nesse experimento é baseada no algoritmo de multiplicação de matrizes proposto por Strassen e apresentado na subseção 2.4.4 do Capítulo 2.

A implementação no WORMS dessa aplicação de teste teve como base a implementação disponibilizada juntamente com a ferramenta Cilk. Porém, com a substituição dos algoritmos de multiplicação para submatrizes pequenas por chamadas à sub-rotina DGEMM de uma biblioteca BLAS. Foi utilizado como critério de parada para a criação de tarefas recursivas o valor de *threshold* de 256 nas implementações para execução em CPU. Já para as implementações que executam em GPU os valores de *threshold* utilizados foram de 1024 como limite inferior e de até 8192 como limite superior para a ordem da submatriz. Os valores de *threshold* para as implementações CPU e GPU das tarefas foram determinados a partir de testes preliminares com cada aplicação.

4.3.1.1 Resultados

Primeiramente foi avaliado o comportamento do WORMS no cenário multi-CPU / mono-GPU da plataforma *Bugio*. A Figura 4.3 apresenta do desempenho em GFlops obtido pelo WORMS na execução de uma multiplicação de matrizes com variação de tamanho de $16x16$ até $8192x8192$. Foram avaliados dois cenários: utilizando somente os núcleos da CPU, ilustrado na Figura 4.3a e utilizando os núcleos da CPU juntamente com a GPU, ilustrado na Figura 4.3b. No cenário CPU + GPU foram utilizados até quatro *TaskProcessors* sendo um associado a uma CPU e a GPU e os demais associados somente a CPUs.

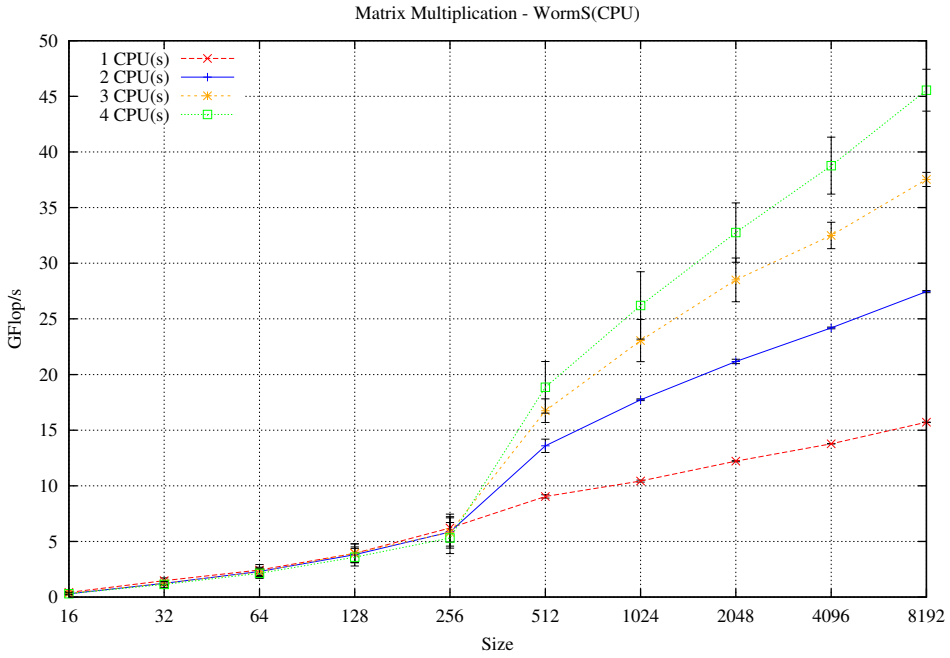
Pela análise dos gráficos pode-se perceber que no cenário somente com CPUs o *speed-up* máximo atingido com o uso de quatro *TaskProcessors* foi levemente inferior a três em relação ao uso de somente um *TaskProcessor*. Já no cenário com CPUs e a GPU observa-se um acréscimo de desempenho da ordem de 110 GFlop/s na comparação com o cenário somente com CPUs. Porém, observa-se também que o acréscimo de *TaskProcessors* associados somente a CPUs acrescentou pouco desempenho em relação ao uso de apenas um *TaskProcessor* associado a uma CPU e a GPU.

Em um segundo momento o comportamento do *middleware* WORMS foi avaliado em um cenário multi-CPU e multi-GPU na plataforma *Newton*. As Figuras 4.4 e 4.5 apresentam o desempenho obtido pelo WORMS em GFlops com a multiplicação de matrizes de ordem 16 até matrizes de ordem 8192. Foram avaliados três cenários: o cenário multi-CPU apresentado no gráfico da Figura 4.4, o cenário multi-CPU / mono-GPU apresentado na Figura 4.5a e o cenário multi-CPU / multi-GPU apresentado na Figura 4.5b.

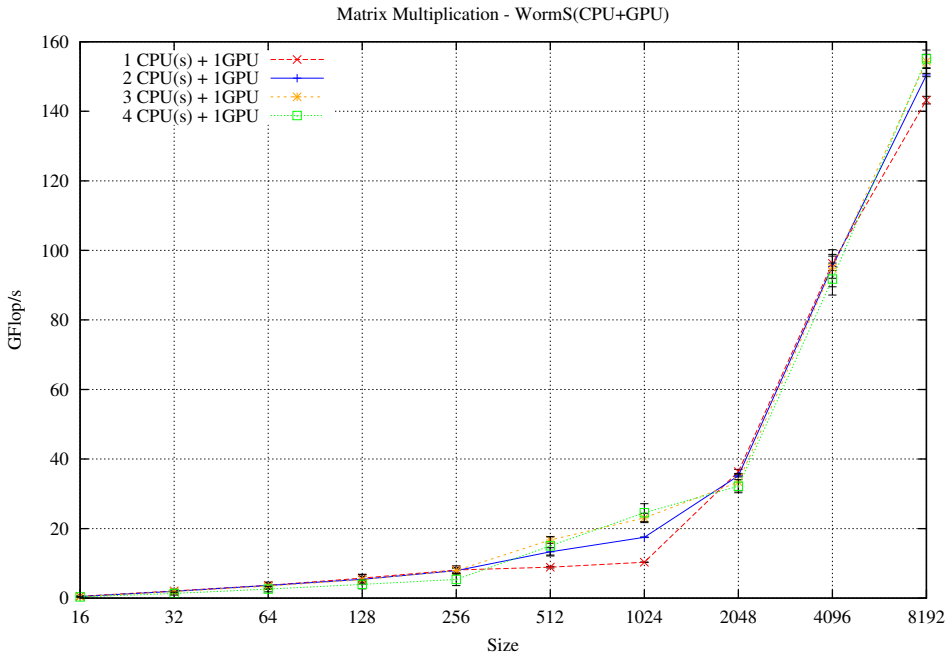
A análise do desempenho apresentado nos gráficos permite observar que no cenário multi-CPU o melhor *speed-up* alcançado com o uso de 10 *TaskProcessors* foi em torno de 6,5. No cenário multi-CPU / mono-GPU, ao acrescentar uma GPU houve aumento de desempenho da ordem de 30 GFlop/s na comparação ao cenário somente com CPUs. Aumento de desempenho semelhante também é observado na comparação entre o uso de 10 *TaskProcessors* (1 *TaskProcessor* CPU + GPU e 9 *TaskProcessors* CPU) em relação ao uso de somente um *TaskProcessor* (CPU + GPU). Já no cenário multi-CPU / multi-GPU, ao acrescentar duas GPUs houve aumento de desempenho de 51 GFlop/s na comparação com execuções somente com CPUs. Na comparação entre o uso de 10 *TaskProcessors* (2 *TaskProcessors* CPU + GPU e 8 *TaskProcessors* CPU) em relação ao uso de 2 *TaskProcessors* (CPU + GPU) observa-se ganho de desempenho de aproximadamente 20 GFlop/s.

4.3.2 Ordenação

A aplicação de teste utilizada nesse experimento é baseada no algoritmo de ordenação *mergesort* apresentado na subseção 2.4.3 do Capítulo 2. Foi utilizado como critério de parada para a criação de tarefas recursivas o valor de *threshold* de 2^{19} como limite inferior



(a) Somente CPUs



(b) CPUs + GPU

Figura 4.3: Desempenho em GFlops do WORMS para a aplicação de teste Multiplicação de Matrizes na plataforma *Bugio*. O valor de *threshold* utilizado é 256.

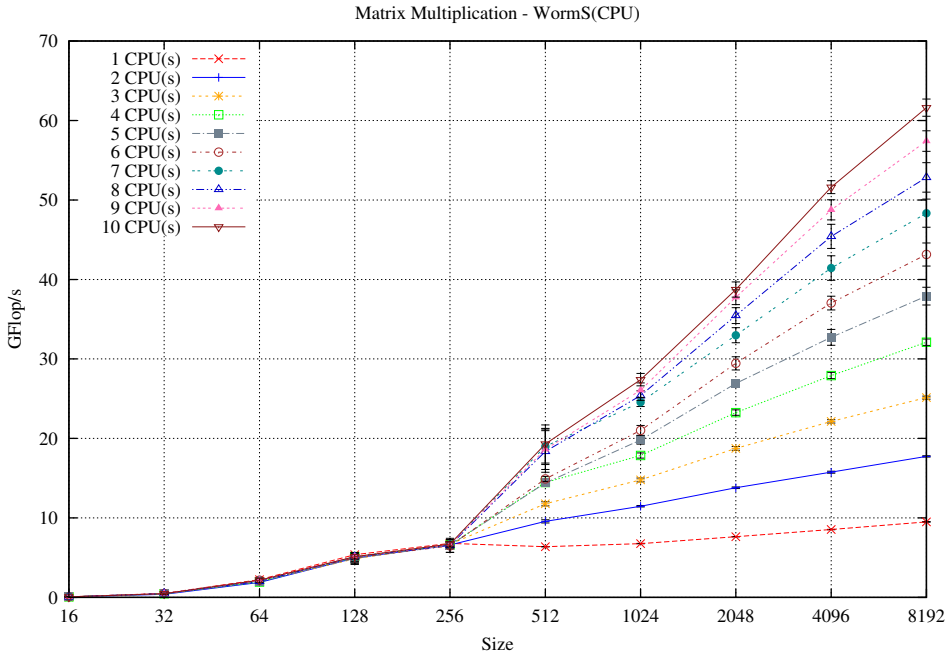


Figura 4.4: Desempenho em GFlops do WORMS para a aplicação de teste Multiplicação de Matrizes utilizando somente CPU na plataforma *Newton*. O valor de *threshold* utilizado é 256.

para o tamanho do subvetor nas implementações para execução em CPU e de até 2^{26} como limite superior para implementações que executam em GPU.

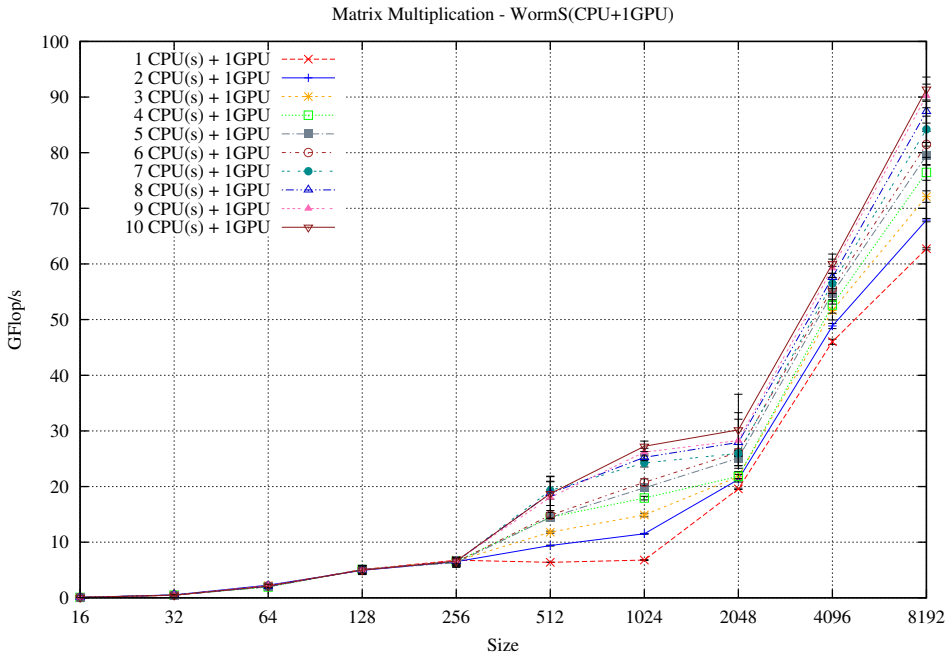
4.3.2.1 Resultados

O comportamento do WORMS com esta aplicação de teste foi avaliado na plataforma *Bugio*. A Figura 4.6 apresenta os tempos de execução obtidos com o WORMS na execução de uma ordenação de elementos inteiros com variação de tamanho dos vetores de 2^{20} até 2^{28}. Foram avaliados dois cenários: utilizando somente os núcleos da CPU (multi-CPU), ilustrado na Figura 4.6a e utilizando os núcleos da CPU juntamente com a GPU (multi-CPU / mono-GPU), ilustrado na Figura 4.6b. No cenário CPU + GPU foram utilizados até quatro *TaskProcessors* sendo um associado a uma CPU e a GPU e os demais associados somente a CPUs.

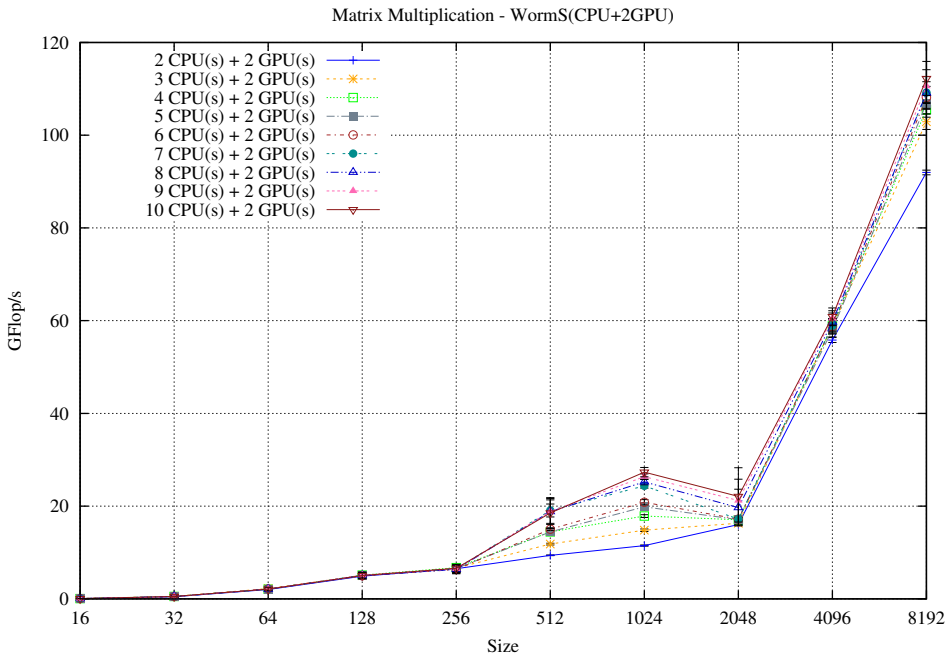
Pela análise dos gráficos pode-se perceber que no cenário somente com CPUs o *speed-up* máximo atingido com o uso de quatro *TaskProcessors* foi aproximadamente 2,75 em relação ao uso de somente um *TaskProcessor*. Já no cenário com CPUs e a GPU observa-se que, apesar do desempenho ser superior em relação ao cenário somente com CPUs, o acréscimo de *TaskProcessors* associados somente a CPUs diminuiu o desempenho em relação ao uso de apenas um *TaskProcessor* associado a uma CPU e a GPU.

4.3.3 Transformação

A aplicação de teste utilizada nesse experimento é baseada no algoritmo recursivo de transformação apresentado na subseção 2.4.2 do Capítulo 2. Foi utilizado como critério de parada para a criação de tarefas recursivas o valor de *threshold* de 2^{16} como limite inferior para o tamanho do subvetor nas implementações para execução em CPU e de até 2^{27} como limite superior para implementações que executam em GPU. No caso da

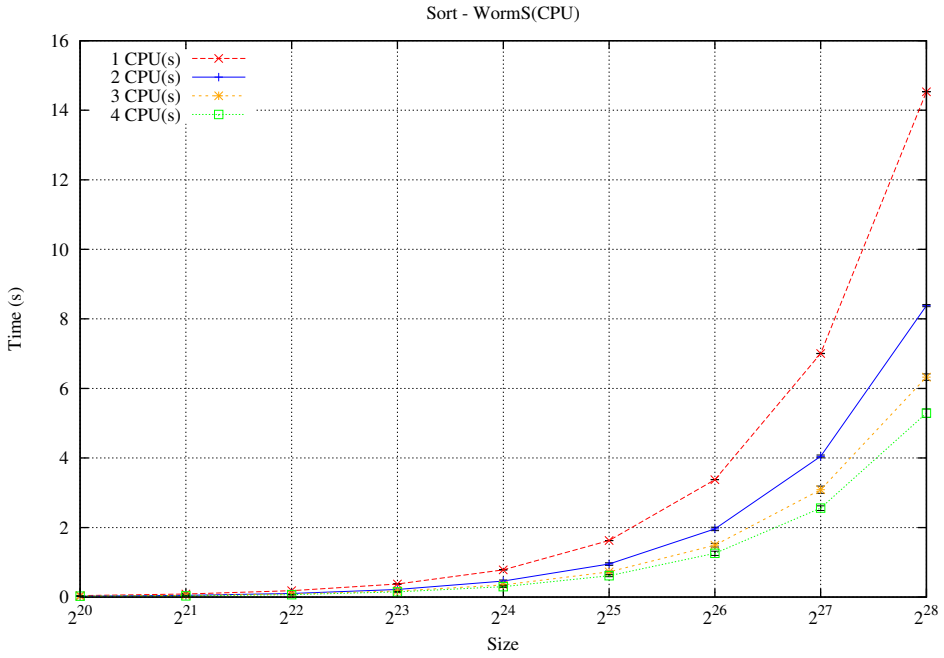


(a) nCPUs + 1GPU

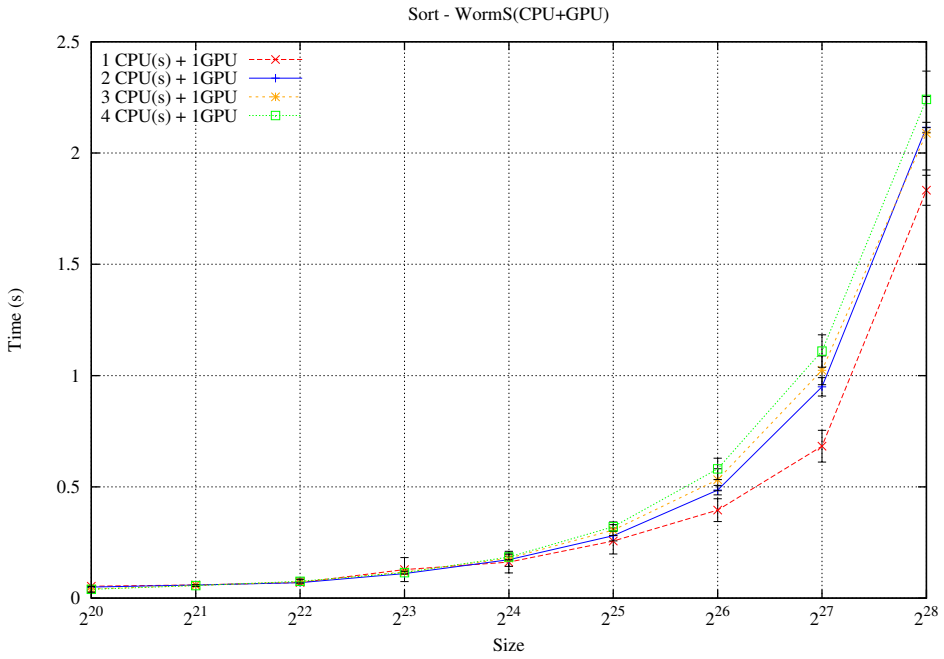


(b) nCPUs + 2GPUs

Figura 4.5: Desempenho em GFlops do WORMS para a aplicação de teste Multiplicação de Matrizes na plataforma *Newton* em cenário multi-CPU e multi-GPU. O valor de *threshold* utilizado é 256.



(a) Somente CPUs



(b) CPUs + GPU

Figura 4.6: Tempo de execução do WORMS para a aplicação de teste Ordenação na plataforma *Bugio*.

implementações GPU, o limiar superior é necessário para evitar que o tamanho máximo de memória disponível na GPU seja ultrapassado. Nesta aplicação, foram considerados dois casos: no primeiro caso a função sintética aplicada nos elementos pelo algoritmo de transformação simula uma carga de computação pequena, já no segundo caso a função sintética possui um laço e, dessa forma, apresenta uma quantidade maior de computação.

4.3.3.1 Resultados

O desempenho apresentado pelo WORMS com esta aplicação de teste foi avaliado na plataforma *Bugio*. A Figura 4.7 apresenta os tempos de execução obtidos na execução de uma transformação de elementos inteiros com variação de tamanho do vetores de 2^{20} até 2^{28} aplicando a função sintética mais leve. Já a Figura 4.8 apresenta os tempos de execução obtidos na execução de uma transformação aplicando a função com maior carga computacional.

Foram avaliados dois cenários: multi-CPU, somente os núcleos da CPU, ilustrado nas Figuras 4.7a e 4.8a e multi-CPU / mono-GPU, utilizando os núcleos da CPU simultaneamente com a GPU, ilustrado nas Figuras 4.7b e 4.8b. No cenário multi-CPU / mono-GPU foram utilizados até quatro *TaskProcessors* sendo um associado a uma CPU e a GPU e os demais associados somente a CPUs.

Considerando o caso da função sintética leve, a análise dos gráficos da Figura 4.7 permite observar que no cenário somente com CPUs o maior *speed-up* foi 3,6 quando utilizados 4 *TaskProcessors*. Já no cenário multi-CPU / mono-GPU, a utilização da GPU em conjunto com os 4 núcleos da CPU provou redução do desempenho.

No caso da função sintética pesada, os gráficos da Figura 4.8 permitem observar que no cenário somente com CPUs o maior *speed-up* alcançado foi 3,75 quando utilizados 4 *TaskProcessors*. No cenário multi-CPU / mono-GPU, a inclusão da GPU em conjunto com os 4 núcleos da CPU ocasionou aumento no desempenho.

4.4 Avaliação da ferramenta de referência Cilk

A ferramenta Cilk, na versão 5.4.6, foi utilizada como referência para o desempenho das aplicações em cenário multi-CPU.

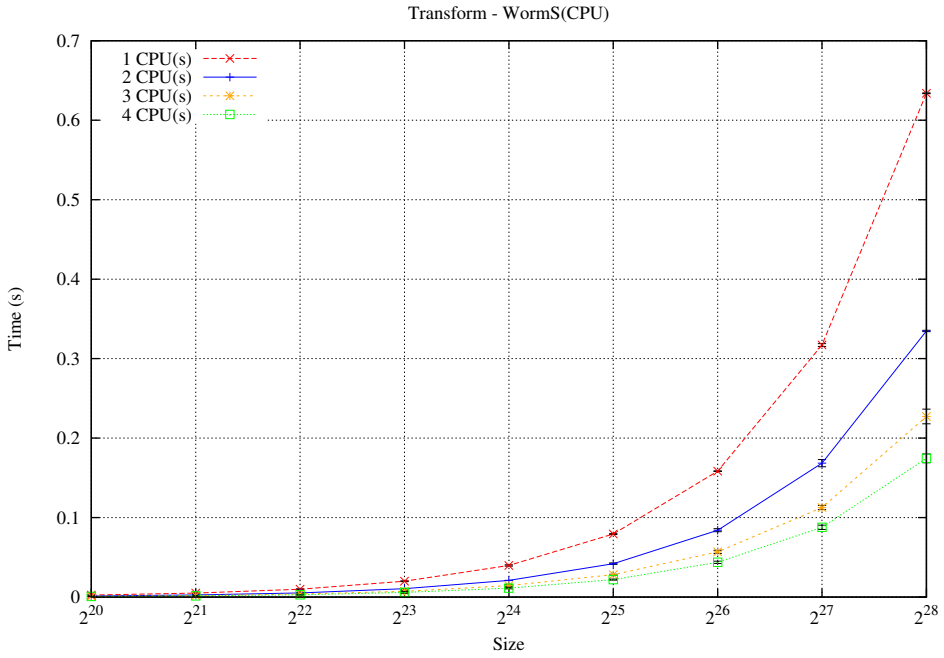
4.4.1 Multiplicação de Matrizes por Algoritmo de Strassen

Essa aplicação de teste é baseada no algoritmo de multiplicação de matrizes proposto por Strassen e apresentado na subseção 2.4.4 do Capítulo 2. A implementação utilizada é uma modificação da versão disponibilizada no pacote do Cilk. As modificações introduzidas consistem na substituição dos algoritmos de multiplicação para submatrizes pequenas por chamadas à sub-rotina DGEMM de uma biblioteca BLAS. O impacto das modificações introduzidas é discutido no experimento da subseção A.1.1 do Apêndice A.

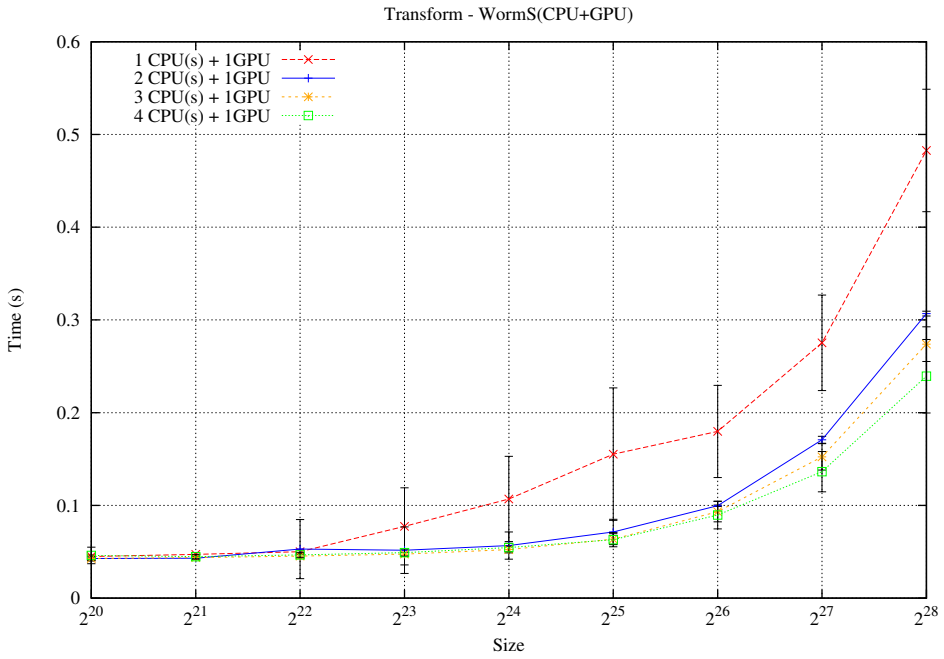
4.4.1.1 Resultados

O comportamento da ferramenta Cilk foi avaliado nas plataformas *Bugio* e *Newton*. O desempenho em GFlops obtido pelo Cilk na execução dessa aplicação é apresentado na Figura 4.9. Foram multiplicadas matrizes quadradas de ordem variando entre 16 e 8192. Foi utilizado o valor 256 como limite inferior para criação recursiva de novas tarefas.

A análise dos gráficos das Figuras 4.9a e 4.9b permite observar que na plataforma *Bugio* o *speed-up* máximo atingido com o uso de 4 núcleos da CPU foi aproximadamente 3.

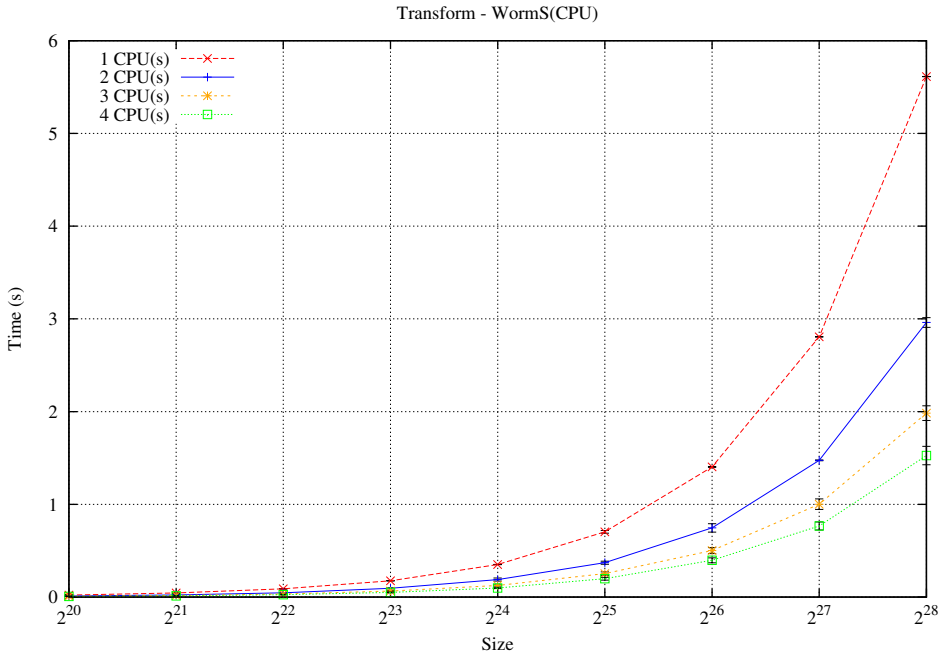


(a) Somente CPUs

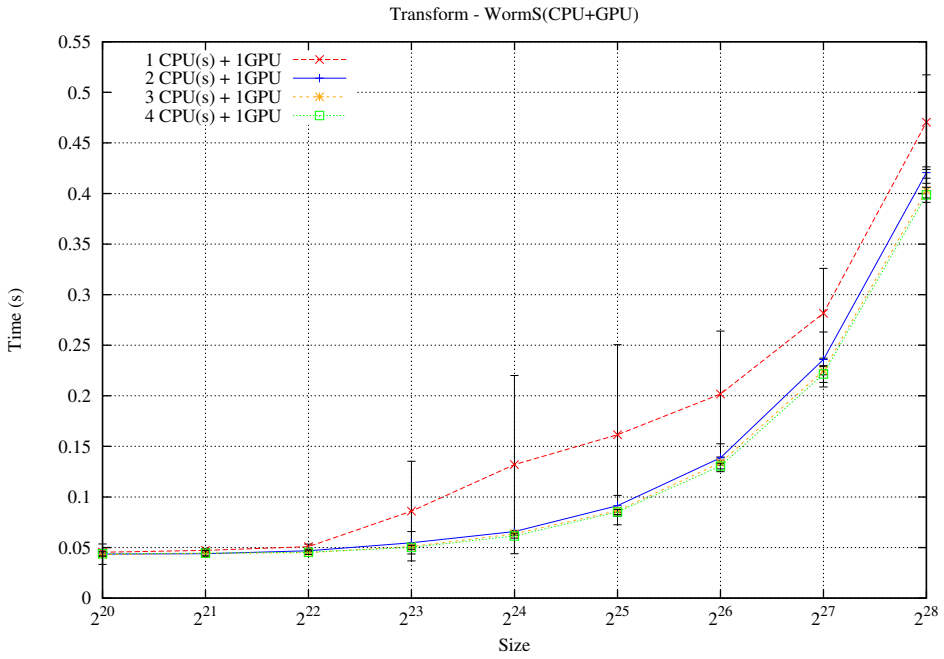


(b) CPUs + GPU

Figura 4.7: Tempo de execução do WORMS para a aplicação de teste Transformação na plataforma *Bugio* utilizando uma função sintética leve.



(a) Somente CPUs



(b) CPUs + GPU

Figura 4.8: Tempo de execução do WORMS para a aplicação de teste Transformação na plataforma *Bugio* utilizando uma função sintética pesada.

Na plataforma *Newton* o *speed-up* máximo foi de aproximadamente 6,5 quando utilizados 10 núcleos da CPU.

4.4.2 Ordenação

Esta aplicação de teste é baseada no algoritmo de ordenação *mergesort* apresentado na subseção 2.4.3 do Capítulo 2. O algoritmo de ordenação da STL (Standard Template Library) foi aplicado nos subvetores de tamanho de entrada menor que o valor de *threshold*.

4.4.2.1 Resultados

O desempenho desta aplicação de teste foi avaliado na plataforma *Bugio*. Nesta plataforma, o algoritmo de ordenação foi aplicado a vetores de entrada com tamanho entre 2^{20} e 2^{28}. Utilizando o valor 2^{19} como limite inferior (*threshold*) para criação recursiva de novas tarefas. A análise do gráfico apresentado na Figura 4.10 possibilita observar que o maior *speed-up* atingido com o uso de quatro núcleos da CPU da plataforma *Bugio* foi aproximadamente 2,6.

4.4.3 Transformação

Esta aplicação de teste é baseada no algoritmo recursivo de transformação apresentado na subseção 2.4.2 do Capítulo 2.

4.4.3.1 Resultados

O desempenho desta aplicação de teste foi avaliado na plataforma *Bugio*. Nesta plataforma, o algoritmo foi aplicado a vetores de entrada com tamanho entre 2^{20} e 2^{28}. Utilizando o valor 2^{16} como limite inferior para criação recursiva de novas tarefas. Assim como na subseção 4.3.3, foram testados casos utilizando duas funções sintéticas: uma leve e uma pesada.

A análise dos gráficos apresentados nas Figuras 4.11 e 4.12 possibilita observar que o maior *speed-up* atingido com o uso de quatro núcleos da CPU da plataforma *Bugio* foi aproximadamente 2,2 no caso da função sintética leve e em torno de 3,6 para o caso da função pesada.

4.5 Avaliação das ferramentas de referência para GPU

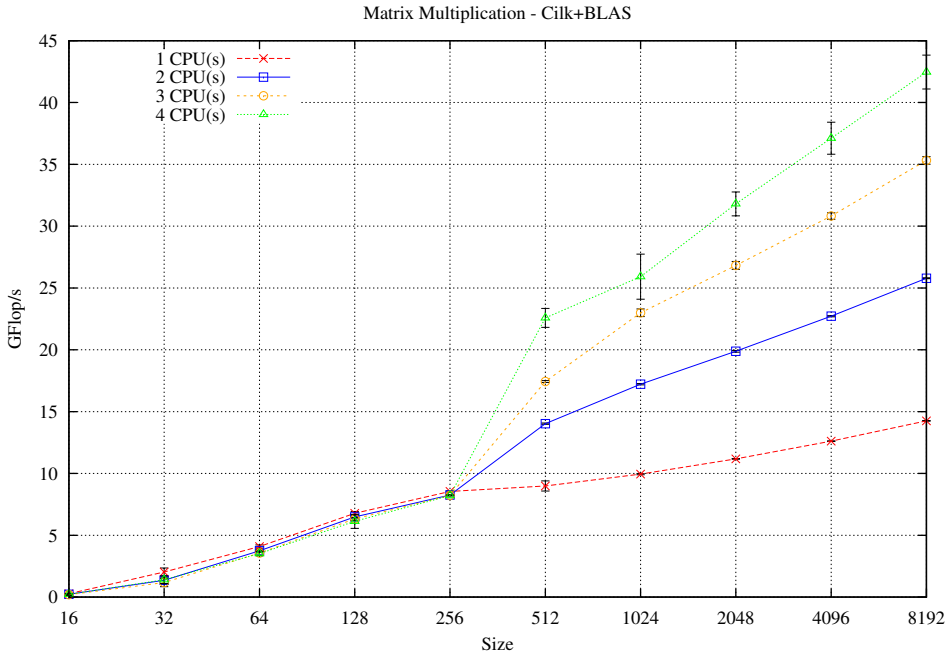
Foram utilizadas como ferramenta de referência para processamento em GPU as ferramentas Thrust (HWU, 2011) e CUBLAS (NVIDIA Developer Zone, 2012b), ambas disponibilizadas juntamente com o ambiente CUDA (NVIDIA Developer Zone, 2012a).

4.5.1 Multiplicação de Matrizes

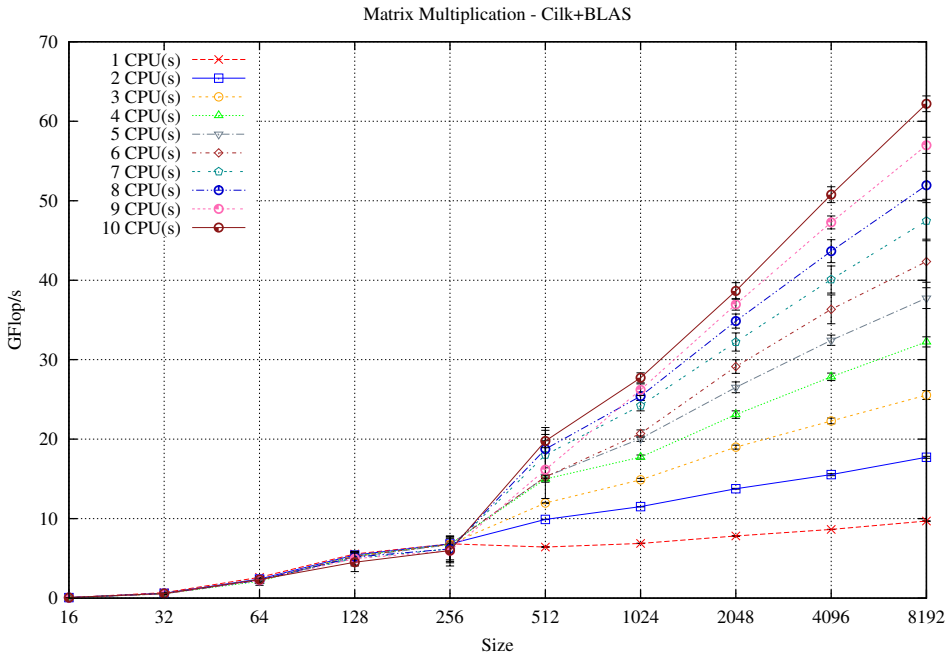
Essa aplicação de teste consiste na execução da sub-rotina de multiplicação de matrizes DGEMM da biblioteca CUBLAS que é uma implementação das sub-rotinas BLAS para execução em GPUs.

4.5.1.1 Resultados

O desempenho desta sub-rotina da biblioteca CUBLAS foi avaliado nas plataformas *Bugio* e *Newton*. Na plataforma *Bugio* foram multiplicadas matrizes com tamanho entre $16x16$ e $4096x4096$. Já na plataforma *Newton* foram utilizados tamanhos entre $16x16$ e $8192x8192$. O tamanho máximo das matrizes foi determinado pela quantidade máxima de memória disponível em cada GPU.



(a) Plataforma *Bugio*



(b) Plataforma *Newton*

Figura 4.9: Desempenho em GFlops do Cilk para a aplicação de teste Multiplicação de Matrizes. O valor de *threshold* utilizado é 256.

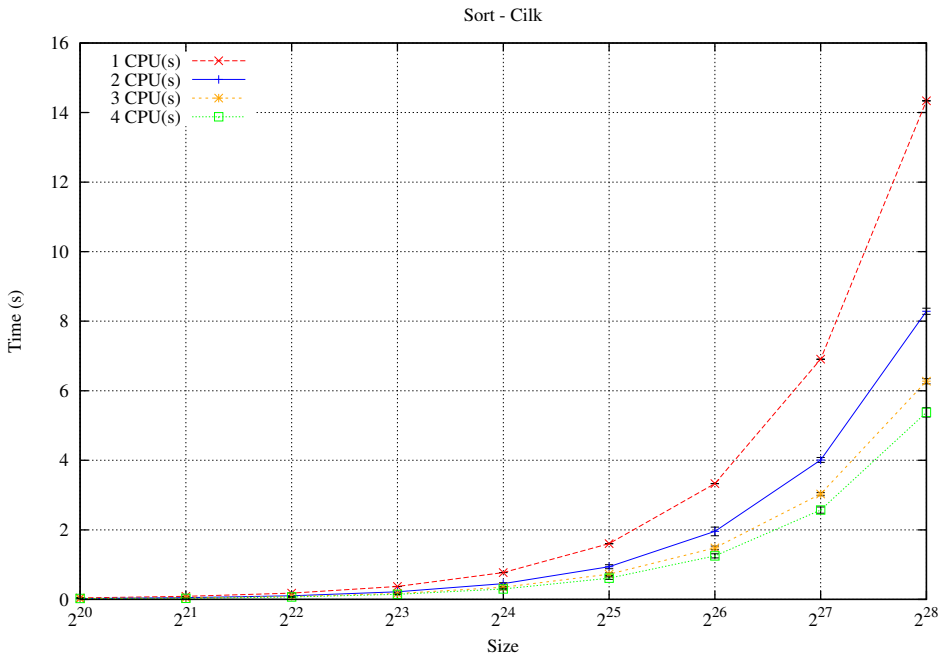


Figura 4.10: Tempo de execução da ferramenta Cilk para a aplicação de teste Ordenação na plataforma *Bugio*.

A Figura 4.13 apresenta o desempenho obtido em GFlops na execução da sub-rotina DGEMM nas duas plataformas de teste.

4.5.2 Ordenação

Esta aplicação de teste consiste na execução do algoritmo de ordenação disponibilizado na ferramenta Thrust.

4.5.2.1 Resultados

O desempenho deste algoritmo foi avaliado na plataforma *Bugio*. Nesta plataforma, o algoritmo foi aplicado a vetores de entrada com tamanho entre 2^{20} e 2^{27}. O tamanho máximo dos vetores de entrada foi determinado pela quantidade de memória disponível na GPU. O gráfico da Figura 4.14 apresenta o tempo de execução do algoritmo de ordenação da ferramenta Thrust na plataforma *Bugio*.

4.5.3 Transformação

Esta aplicação de teste consiste na execução do algoritmo de transformação disponibilizado na ferramenta Thrust.

4.5.3.1 Resultados

O desempenho deste algoritmo foi avaliado na plataforma *Bugio*. Nesta plataforma, o algoritmo foi aplicado a vetores de entrada com tamanho entre 2^{20} e 2^{28}. O tamanho máximo dos vetores de entrada foi determinado pela quantidade de memória disponível na GPU. Assim como na subseção 4.3.3, foram testados casos utilizando duas funções sintéticas.

Os gráficos das Figuras 4.15 e 4.16 apresentam o tempo de execução do algoritmo de transformação da ferramenta Thrust na plataforma *Bugio*.

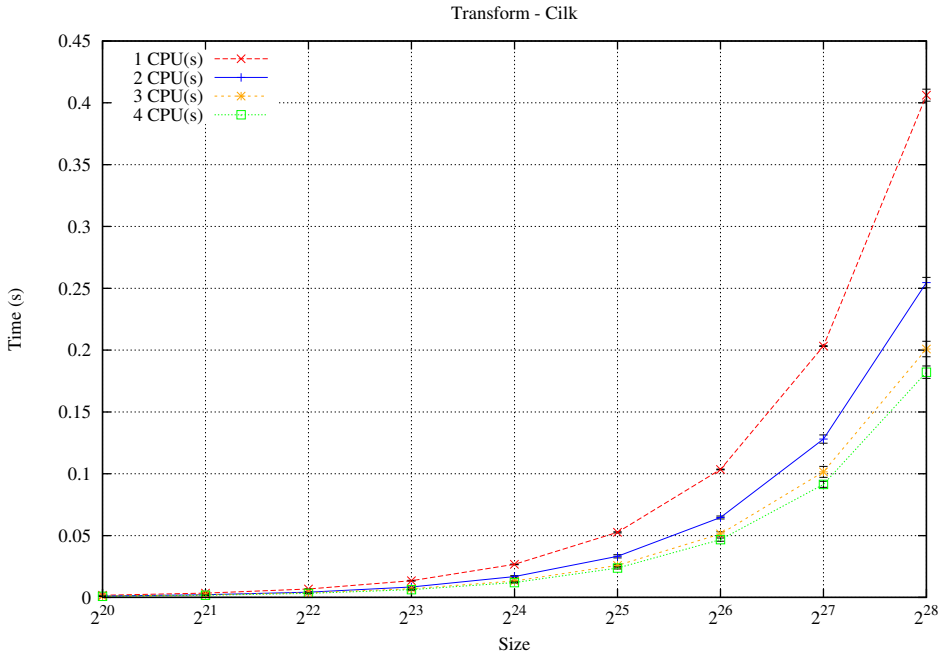


Figura 4.11: Tempo de execução da ferramenta Cilk para a aplicação de teste Transformação na plataforma *Bugio* utilizando uma função sintética leve.

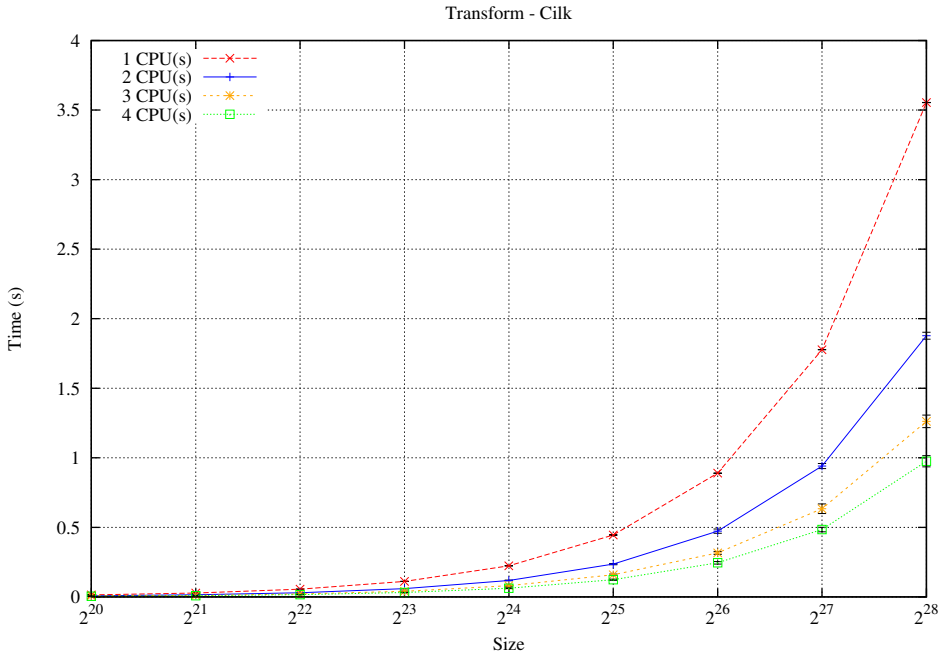
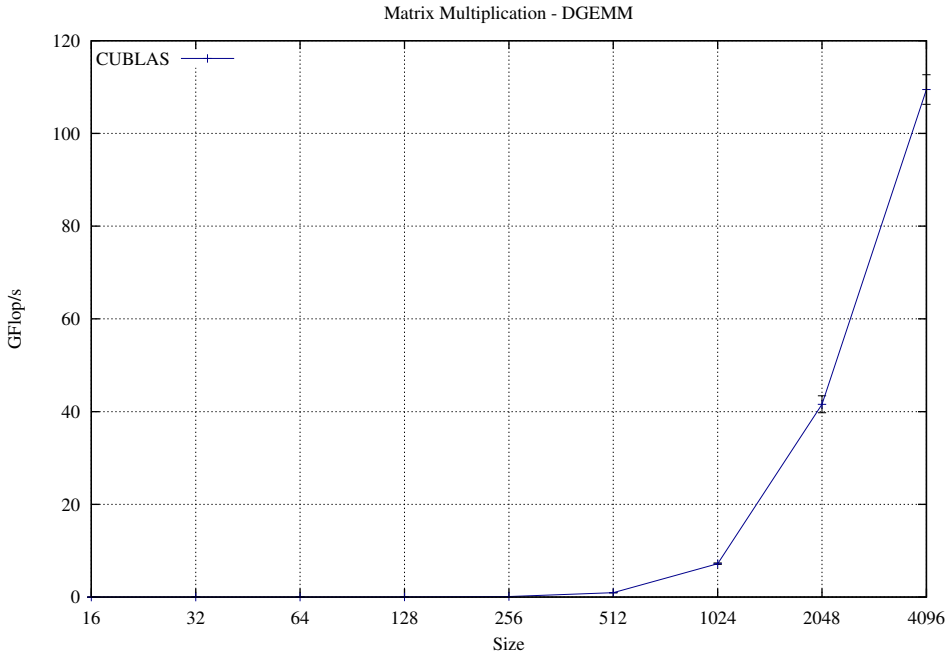


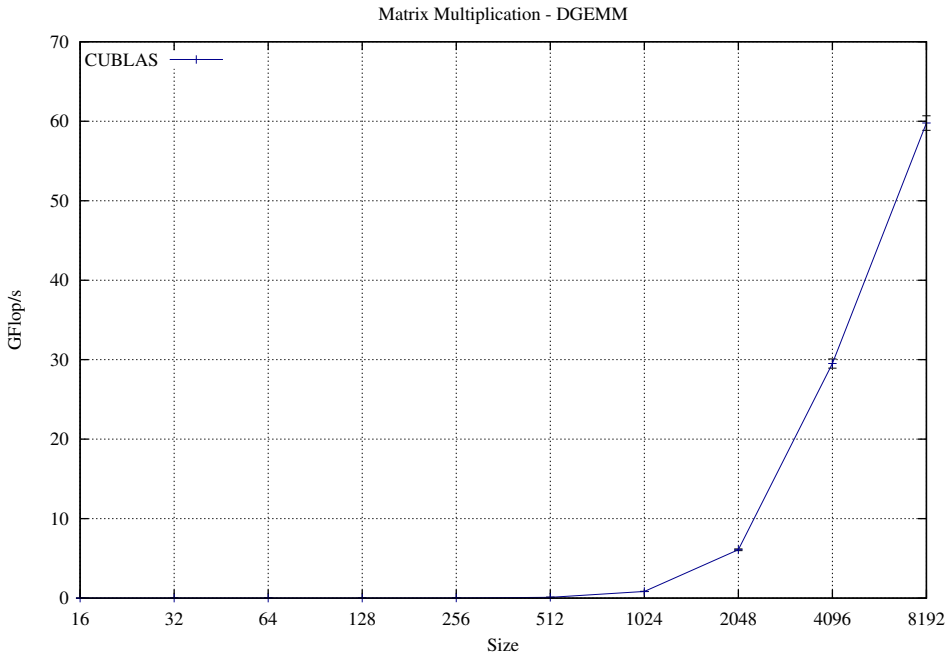
Figura 4.12: Tempo de execução da ferramenta Cilk para a aplicação de teste Transformação na plataforma *Bugio* utilizando uma função sintética pesada.

4.6 Análise comparativa dos resultados do WORMS com ferramentas de referência

Nesta seção é apresentada uma avaliação comparativa entre os resultados obtidos com o *middleware* WORMS e os resultados apresentados pelas ferramentas de referência. A comparação considera o melhor desempenho obtido pelo Cilk, pela ferramenta para GPU



(a) Plataforma *Bugio*



(b) Plataforma *Newton*

Figura 4.13: Desempenho em GFlops da biblioteca CUBLAS para a aplicação de teste Multiplicação de Matrizes.

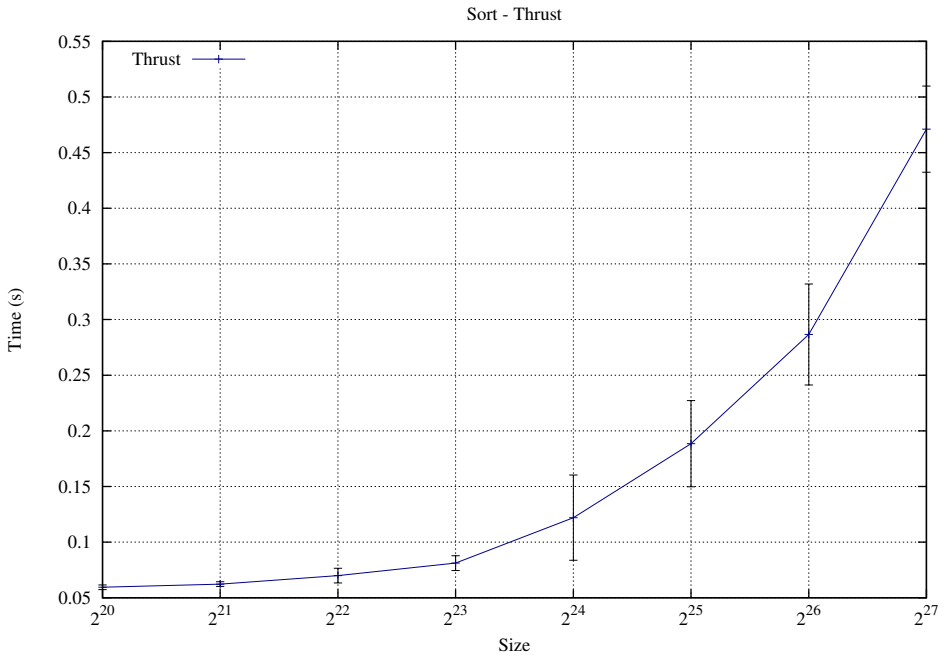


Figura 4.14: Tempo de execução da ferramenta Thrust para a aplicação de teste Ordenação na plataforma *Bugio*.

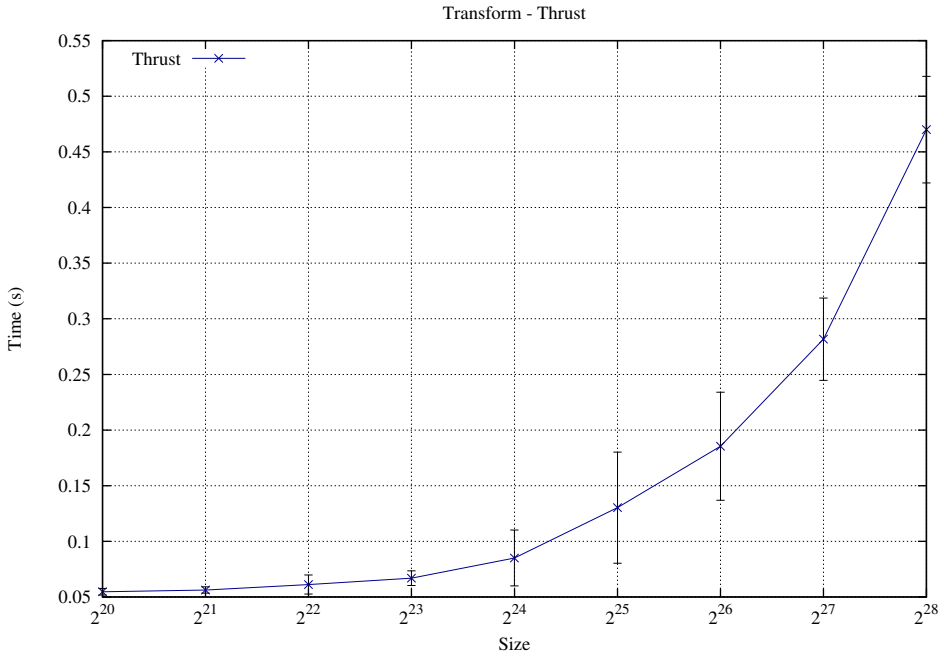


Figura 4.15: Tempo de execução da ferramenta Thrust para a aplicação de teste Transformação utilizando uma função sintética leve na plataforma *Bugio*.

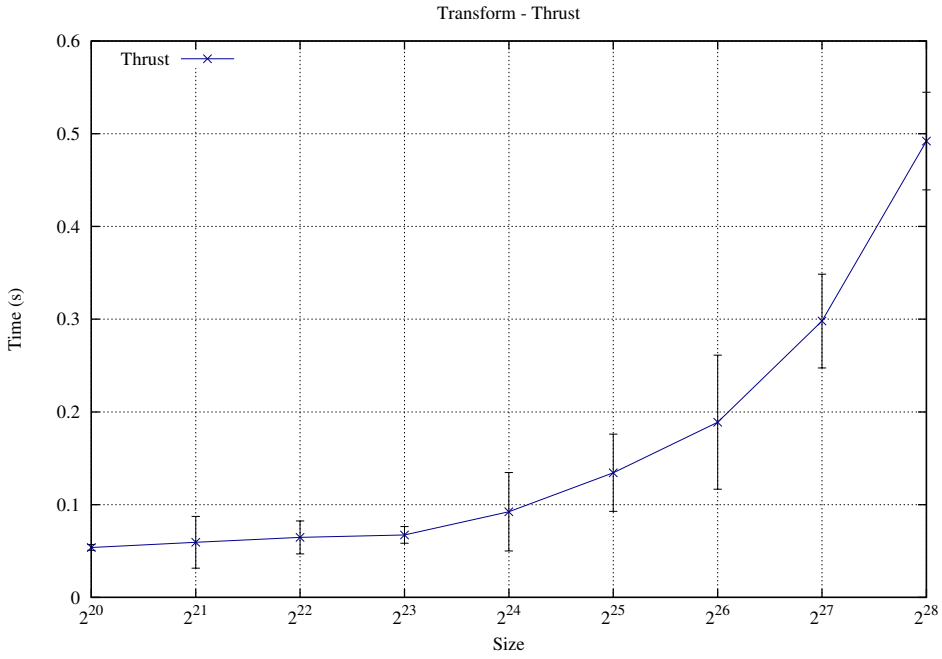


Figura 4.16: Tempo de execução da ferramenta Thrust para a aplicação de teste Transformação utilizando uma função sintética pesada na plataforma *Bugio*.

(Thrust ou CUBLAS) e por três configurações do WORMS (multi-CPU, multi-CPU / mono-GPU e multi-CPU / multi-GPU).

4.6.1 Multiplicação de Matrizes

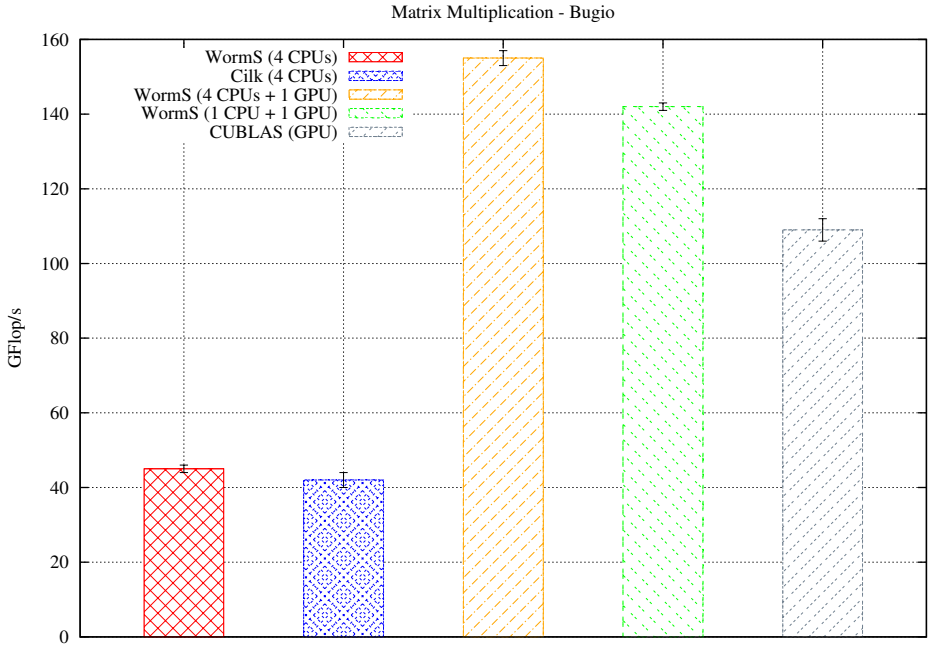
Os gráficos da Figura 4.17 e a Tabela 4.1 apresentam de forma resumida a comparação do desempenho de pico da aplicação multiplicação de matrizes entre as diferentes ferramentas. Os resultados tabulados nessa tabela foram apresentados individualmente de forma detalhada nos gráficos das Figuras 4.3, 4.4, 4.5, 4.9, 4.13.

A partir dos resultados apresentados pode-se concluir que o *middleware* WORMS utilizando núcleos da CPU e as GPUs simultaneamente obteve desempenho superior tanto ao da ferramenta Cilk quanto da biblioteca CUBLAS nas duas plataformas de teste. Além disso, na configuração utilizando somente CPUs o desempenho do WORMS foi equivalente ao da ferramenta Cilk.

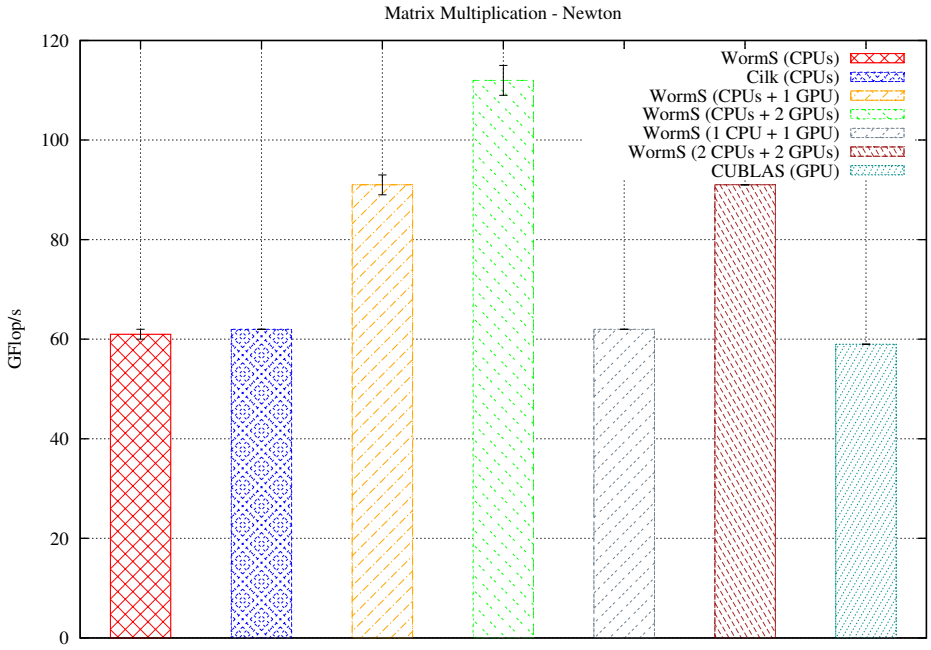
4.6.2 Ordenação

O gráfico da Figura 4.18 apresenta de forma resumida a comparação do desempenho da aplicação ordenação entre as diferentes ferramentas. Os resultados agrupados nesse gráfico foram apresentados individualmente nos gráficos das Figuras 4.6, 4.10, 4.14.

Os resultados apresentados permitem concluir que o *middleware* WORMS utilizando núcleos da CPU e as GPUs simultaneamente apresentou desempenho superior ao da ferramenta Cilk e ao WORMS utilizando somente CPUs. Além disso, na configuração utilizando somente CPUs o desempenho do Cilk e do WORMS foi equivalente. Porém, na comparação com os resultados da ferramenta Thrust observa-se que o desempenho do WORMS foi inferior e que o acréscimo de *TaskProcessors* associados somente a CPUs piorou o desempenho em relação ao uso de somente um *TaskProcessor* associado a apenas uma CPU e a GPU. Nesse cenário, um aspecto positivo é que o WORMS, devido à seu



(a) Plataforma *Bugio*



(b) Plataforma *Newton*

Figura 4.17: Comparação do desempenho de pico da aplicação Multiplicação de Matrizes no *middleware* WORMS em relação à ferramentas de referência para CPU (Cilk) e para GPU (CUBLAS).

	Gflop/s ($\overline{x}$)	**(σ)**	**Max 1**	**Max 2**
WORMS (CPUs)	45,55264382	1,88435444	8192	8192
WORMS (CPUs + 1 GPU)	155,07241284	2,54728217	8192	8192
WORMS (1 CPU + 1 GPU)	143,20779802	1,08916699	8192	8192
Cilk (CPUs)	42,76373233	2,39604088	8192	8192
CUBLAS (GPU)	109,46772692	3,19048073	4096	4096

(a) Plataforma *Bugio*

	Gflop/s ($\overline{x}$)	**(σ)**	**Max 1**	**Max 2**
WORMS (CPUs)	61,61670410	1,08039834	8192	8192
WORMS (CPUs + 1 GPU)	91,42590800	2,18382777	8192	8192
WORMS (CPUs + 2 GPUs)	112,23507442	3,67123881	8192	8192
WORMS (1 CPU + 1 GPU)	62,73296850	0,17874747	8192	8192
WORMS (2 CPUs + 2 GPUs)	91,96576156	0,49536659	8192	8192
Cilk (CPUs)	62,19400990	0,99215960	8192	8129
CUBLAS (GPU)	59,77373222	0,91834399	8192	8192

(b) Plataforma *Newton*

Tabela 4.1: Comparação do desempenho de pico da aplicação Multiplicação de Matrizes no *middleware* WORMS em relação à ferramentas de referência para CPU (Cilk) e para GPU (CUBLAS). A coluna *Gflop/s* apresenta a média do desempenho de pico obtido em elementos ordenados por segundo. A coluna *(σ)* apresenta o desvio padrão dos dados da coluna *Gflop/s*. A coluna *Max 1* indica o tamanho máximo de entrada testado. A coluna *Max 2* indica o tamanho máximo de entrada testado quando obtido o desempenho de pico.

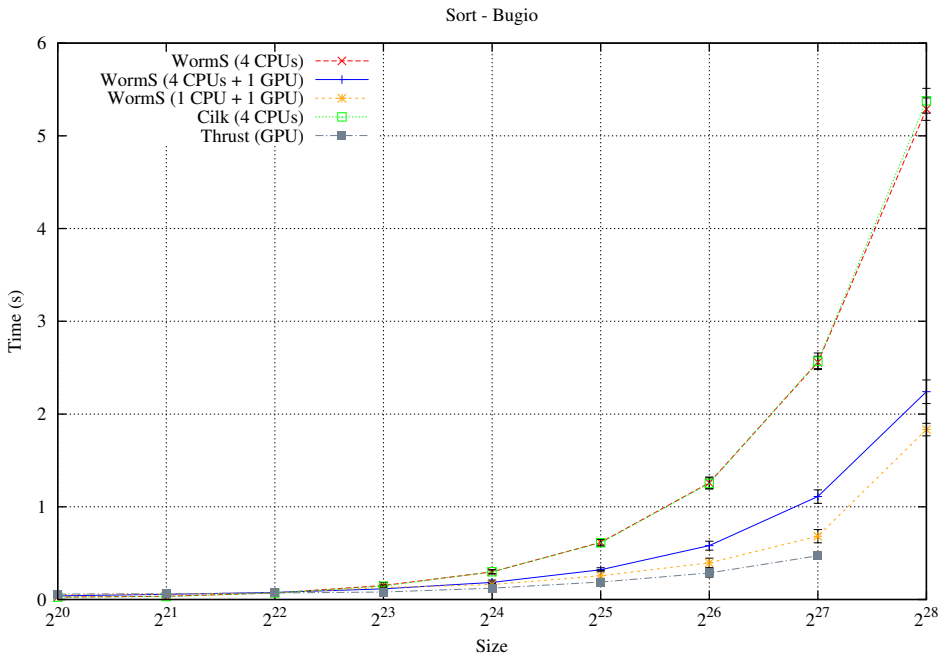


Figura 4.18: Comparação do desempenho da aplicação Ordenação no *middleware* WORMS em relação à ferramentas de referência para CPU (Cilk) e para GPU (Thrust).

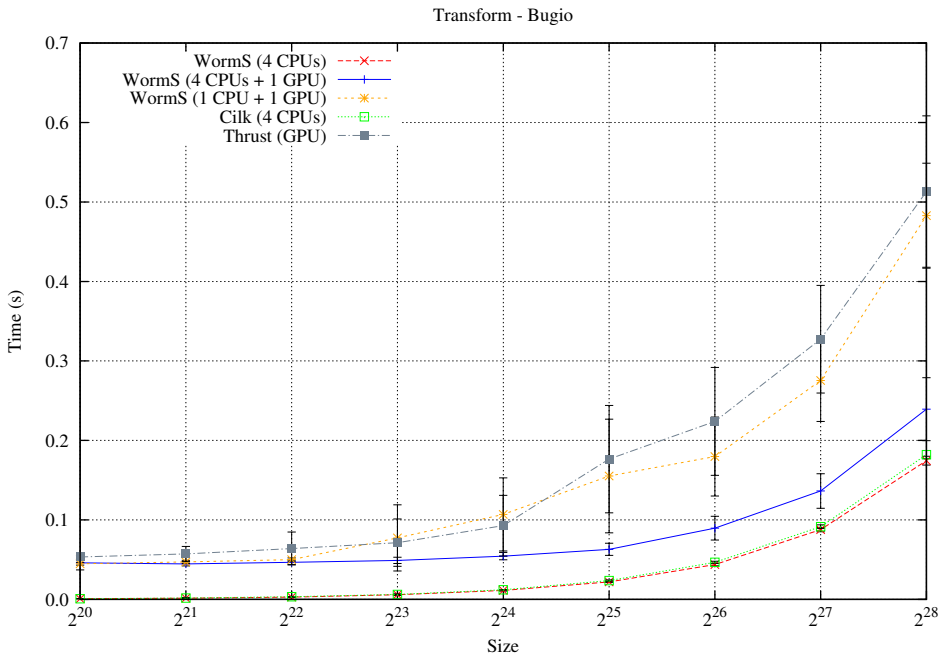


Figura 4.19: Comparação do desempenho da aplicação Transformação no *middleware* WORMS em relação à ferramentas de referência para CPU (Cilk) e para GPU (Thrust) utilizando uma função sintética leve.

paradigma de programação, permitiu a ordenação de vetores de tamanho maior do que o permitido pela ferramenta Thrust.

4.6.3 Transformação

Os gráficos das Figuras 4.19 e 4.20 apresentam de forma resumida a comparação do desempenho da aplicação transformação entre as diferentes ferramentas. Os resultados agrupados nesse gráfico foram apresentados individualmente nas Figuras 4.7, 4.8, 4.11, 4.12, 4.15 e 4.16.

A partir dos resultados obtidos pode-se concluir que, no caso dos testes com a função sintética leve, o WORMS utilizando núcleos da CPU e as GPUs simultaneamente apresentou desempenho inferior ao da ferramenta Cilk e ao do WORMS utilizando somente CPUs. Já no caso dos testes com a função sintética pesada, ocorreu o oposto, ou seja, o desempenho do WORMS ao utilizar as CPUs e a GPU simultaneamente foi superior tanto ao do Cilk quanto ao do Thrust.

Para essa aplicação, o processamento em GPU (Thrust) utilizando tanto a função sintética leve quanto a pesada apresenta praticamente o mesmo desempenho. Isso ocorre porque a maior parte do tempo de execução na GPU é gasto nas transferências de memória entre a CPU e a GPU, fazendo com que o uso de uma função mais pesada na transformação não tenha impacto significativo no desempenho.

A inclusão da GPU nos testes com o WORMS que no caso do teste com a função leve ocasionou perda de desempenho, teve efeito contrário nos testes com utilização da função pesada.

Quando utilizada a função sintética leve, a operação aplicada em cada elemento pelo algoritmo de transformação é uma operação muito simples e o custo de transferir os dados de e para a GPU não é compensado pelo maior desempenho na execução desta operação, tornando desvantajoso executar o algoritmo em GPU. Já quando se usa a função sintética pesada, o desempenho da GPU ao executar a aplicação permanece o mesmo, porém o

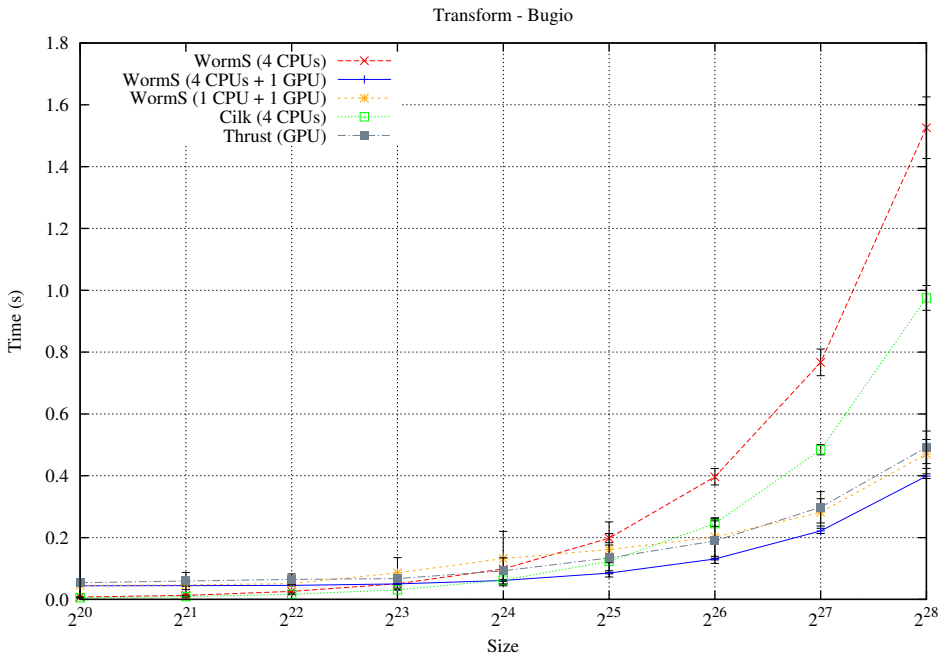


Figura 4.20: Comparação do desempenho da aplicação Transformação no *middleware* WORMS em relação à ferramentas de referência para CPU (Cilk) e para GPU (Thrust) utilizando uma função sintética pesada.

desempenho da CPU diminui bastante, tornando vantajoso executar a aplicação nos dois recursos, conforme mostrado no gráfico anterior.

5 CONSIDERAÇÕES FINAIS

A incorporação de processadores *multicore* e de aceleradores aos sistemas de processamento de alto desempenho tem mostrado que os modelos e ferramentas atuais para programação paralela não são adequados para cenários híbridos, podendo gerar aplicações pouco portáveis. O paralelismo de tarefas que tem sido apontado como um paradigma de programação genérico e expressivo pode ser adotado neste cenário. Entretanto, o uso eficiente do paralelismo de tarefas depende da adoção de algoritmos de escalonamento e balanceamento de carga dinâmicos como o algoritmo de roubo de tarefas.

Neste contexto, este trabalho teve por objetivo explorar e validar o escalonamento baseado na técnica de roubo de tarefas em sistemas paralelos híbridos compostos de CPUs *multicore* e de GPUs, que foram empregadas como aceleradores no cálculo de tarefas de propósito geral. Esta proposta foi concretizada no *middleware* WORMS que permite a execução de tarefas paralelas em sistemas multi-CPU e multi-GPU escalonando essas tarefas através de um algoritmo de roubo de tarefas. No WORMS cada tarefa pode possuir tanto implementações para execução em CPUs quanto em GPUs. Além disso, durante a sua execução uma tarefa pode criar novas tarefas dinamicamente. Com isso, pode-se atingir um aproveitamento mais eficiente do potencial de paralelismo presente na arquitetura computacional híbrida, além de tornar a aplicação mais portável.

Os resultados obtidos na avaliação experimental permitiram constatar ser possível superar, em alguns casos, o desempenho tanto das ferramentas para execução em CPU quanto para execução em GPU ao se utilizar simultaneamente os recursos CPU e GPU. Esse comportamento foi observado em aplicações com carga computacional mais intensa, como a multiplicação de matrizes e um dos casos testados da transformação. Com algumas aplicações, porém, os resultados mostraram ser mais eficiente utilizar apenas a GPU ou as CPUs, em particular nos casos em que a diferença de desempenho entre os recursos e/ou o uso intenso de memória torna mais vantajoso executar a aplicação totalmente em um ou outro recurso, como no caso da aplicação ordenação.

5.1 Contribuições

A principal contribuição deste trabalho foi demonstrar que a técnica de escalonamento por roubo de tarefas pode apresentar bom desempenho em algumas aplicações quando aplicada em arquiteturas computacionais híbridas. As demais contribuições alcançadas neste trabalho foram:

- Apresentar o paradigma de paralelismo de tarefas, suas implicações e exemplos de ferramentas onde é implementado;

- Apresentar o escalonamento por roubo de tarefas, as ferramentas que o utilizam e os diversos cenários computacionais onde já foi aplicado;
- Oferecer um *middleware* que permite utilizar o paralelismo de tarefas e o escalonamento de tarefas em arquiteturas paralelas multi-CPU e multi-GPU;
- Avaliar o *middleware* implementado com três aplicações de teste e compará-lo com ferramentas de referência para programação em GPUs e para uso de paralelismo de tarefas em CPUs *multicore*.

Foram escritos um artigo e dois resumos para eventos nacionais e regionais durante o desenvolvimento desta dissertação. Foi submetido e aprovado um artigo para o evento nacional da área WSCAD-SSC 2012 (XIII Simpósio em Sistemas Computacionais) (PINTO; MAILLARD, 2012a). Também foram publicados resumos sobre o trabalho nos eventos regionais ERAD 2012 (XII Escola Regional de Alto Desempenho) (PINTO et al., 2012) e WSPPD 2012 (X Workshop de Processamento Paralelo e Distribuído) (PINTO; MAILLARD, 2012b).

5.2 Trabalhos Futuros

Em continuidade ao trabalho desenvolvido, os trabalhos futuros preveem a redução das limitações técnicas de implementação identificadas na versão atual e a execução de testes em arquiteturas com um número maior de GPUs.

Pretende-se também ampliar a avaliação experimental realizada incluindo novas aplicações de teste e novas ferramentas de referência, em especial ferramentas que também façam o uso simultâneo de CPUs e GPUs mesmo que fora do cenário de paralelismo de tarefas e escalonamento por roubo de tarefas.

Outros trabalhos futuros que podem ser realizados a partir desta dissertação são: a implementação do algoritmo de roubo de tarefas do WORMS em ferramentas com paralelismo de tarefas que ofereçam suporte à memória compartilhada distribuída, inclusão de variações no algoritmo de roubo de tarefas do WORMS e a inclusão do suporte a arquiteturas com memória distribuída.

APÊNDICE A EXPERIMENTOS COMPLEMENTARES

Neste apêndice são apresentados resultados experimentais complementares. Esses resultados foram utilizados de forma complementar para determinação de parâmetros e conferência de resultados dos testes apresentados no Capítulo 4. Além disso, alguns desses testes também foram utilizados para definir aspectos da implementação das aplicações de teste e do *middleware* WORMS.

A.1 Experimentos com Cilk

A.1.1 Comparação da implementação original do algoritmo de Strassen com a implementação com uso de uma biblioteca BLAS

O experimento descrito nesta subseção teve por objetivo comparar o desempenho entre a implementação original da multiplicação de matrizes por meio do algoritmo de Strassen disponível no Cilk 5.4.6 e a nova implementação utilizada neste trabalho, onde foram adicionadas chamadas a uma biblioteca BLAS ao código da implementação original.

A implementação original, cujo desempenho é mostrado no gráfico da Figura A.1, utiliza três algoritmos diferentes para multiplicação de matrizes:

- uma versão otimizada do algoritmo de Strassen propriamente dito utilizada para matrizes de tamanhos grandes;
- uma versão baseada em divisão e conquista utilizada para matrizes de tamanhos médios a grandes;
- uma versão otimizada do algoritmo ingênuo utilizada para matrizes de tamanhos pequenos a médios[1].

Na nova implementação realizada para este trabalho foi mantido apenas a versão otimizada do algoritmo de Strassen, sendo as versões baseadas em divisão e conquista e no algoritmo ingênuo otimizados substituídas por chamadas à sub-rotina DGEMM (*Double-precision General Matrix Multiply*) de uma biblioteca BLAS. O desempenho dessa implementação é apresentado no gráfico da Figura A.2.

A comparação do desempenho apresentado nos gráficos das figuras A.1 e A.2 mostra que a versão que utiliza chamadas BLAS é mais rápida que a versão original para tamanhos suficientemente grandes. Conforme mostrado na Tabela A.1 o desempenho de pico

[1] No código disponibilizado no Cilk 5.4.6, os valores de *threshold* usados são: 64 (troca do algoritmo de Strassen para o algoritmo de divisão e conquista) e 16 (troca do algoritmo de divisão e conquista para o algoritmo ingênuo otimizado).

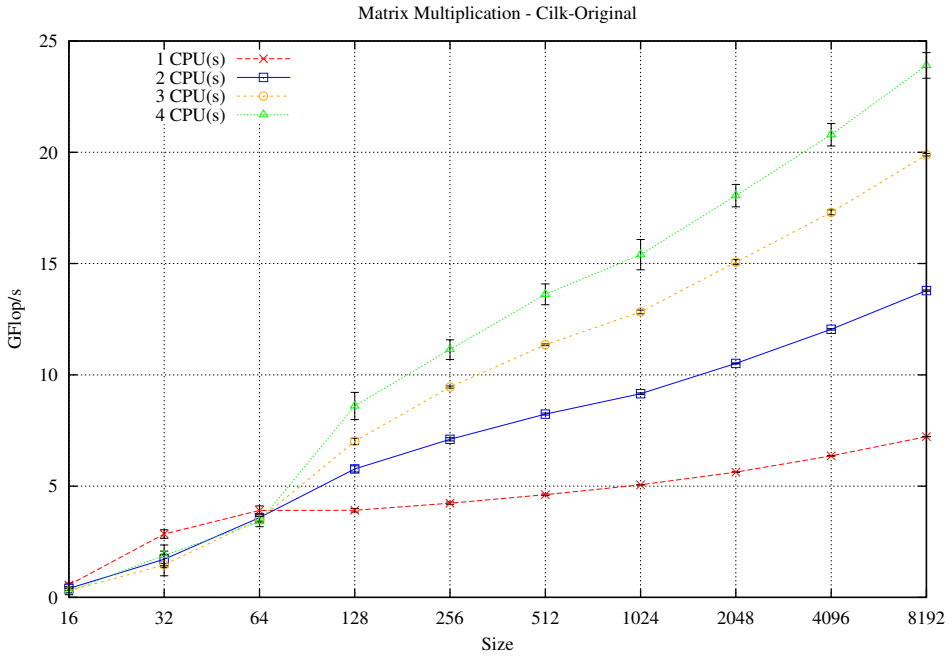


Figura A.1: Desempenho do Cilk em GFlops com o algoritmo de Strassen original do Cilk 5.4.6

da implementação com chamadas BLAS é em torno de 77,6% maior que na implementação original para o maior tamanho de matriz testado utilizando 4 CPUs. Dessa forma essa foi a implementação utilizada na comparação com o *middleware* WORMS.

A.1.2 Comparação do desempenho do Cilk 5 com o Intel Cilk Plus

A versão do Cilk utilizada nos testes apresentados neste trabalho é a 5.4.6, a qual é a última versão oficial disponibilizada pelo MIT. Optou-se por utilizar o Cilk 5.4.6 porque essa é a última versão utilizada nos artigos que descrevem a implementação do Cilk ou que propõe modificações nesta (FRIGO; LEISERSON; RANDALL, 1998; BENDER; RABIN, 2000; LEISERSON, 2009).

A tecnologia do Cilk foi adquirida pela Intel que distribui as versões atuais do Cilk com o nome de Intel Cilk Plus. Recentemente, a Intel disponibilizou uma versão de código aberto do Intel Cilk Plus, que até então estava disponível apenas como *software* comercial. Essa versão está disponível como um *branch* não oficial do compilador GCC (CORPORATION, 2013).

Esta subseção apresenta uma comparação entre o desempenho do Cilk 5.4.6 utilizado neste trabalho e do Intel Cilk Plus (através do *branch* do GCC). Conforme pode ser observado no gráfico da Figura A.3 o desempenho do Cilk 5.4.6 é ligeiramente inferior ao do Intel Cilk Plus. Esse teste foi realizado com o algoritmo para multiplicação de matrizes de Strassen (utilizando chamadas BLAS). Conforme mostrado na Tabela A.2 o Intel Cilk Plus é cerca 3,4% mais rápido que o do Cilk 5.4.6 para o maior tamanho de matriz testado.

A.2 Experimentos com CUBLAS

Os resultados desta subseção mostram o desempenho da biblioteca CUBLAS obtido na execução de uma chamada BLAS do tipo DGEMM incluindo ou excluindo o tempo

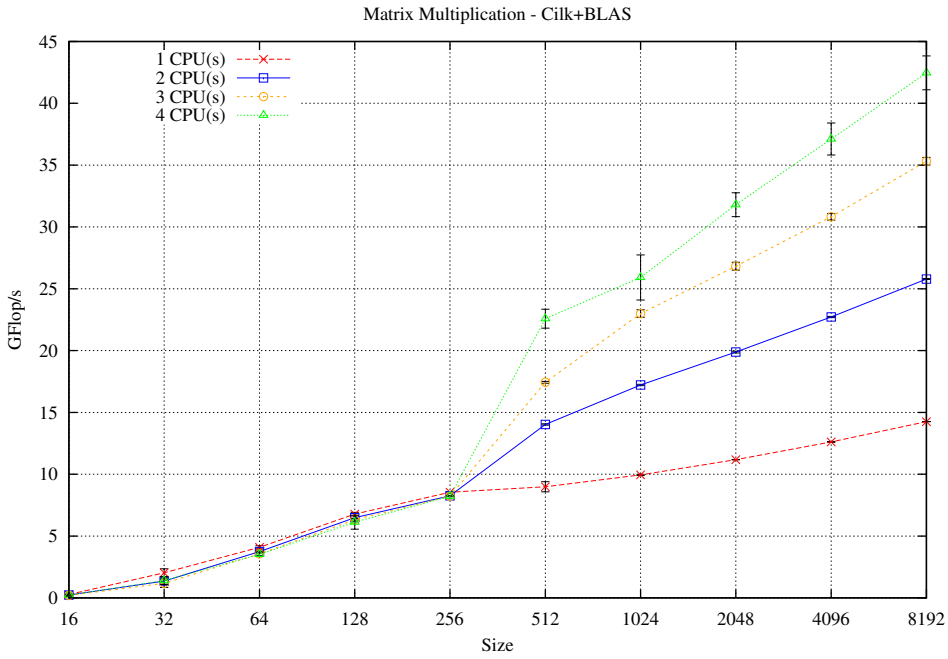


Figura A.2: Desempenho do Cilk em GFlops com o algoritmo de Strassen utilizando chamadas à biblioteca BLAS.

Ordem da Matriz (n)	**Diferença em GFlops**	**Diferença em %**
16	-0,0458723	-16,6637490524
32	-0,48567163	-25,7187444048
64	0,12072091	3,548764914
128	-2,47467202	-28,7713425083
256	-2,92489075	-26,268338338
512	8,9617104	65,8032213009
1024	10,51327111	68,2676751802
2048	13,74766117	76,1457583291
4096	16,32217936	78,509000876
8192	18,55856835	77,6401806392

Tabela A.1: Diferença no desempenho de pico com 4 CPUs entre a implementação original do algoritmo de multiplicação de matrizes de Strassen e a implementação utilizando chamadas BLAS. A primeira coluna indica a ordem das matrizes multiplicadas. A coluna **Diferença em GFlops** apresenta a diferença entre o desempenho da implementação utilizando chamadas BLAS e a implementação original disponível nos exemplos do Cilk 5.4.6. A coluna **Diferença em %** mostra em % a diferença de desempenho entre a implementação com BLAS e a original.

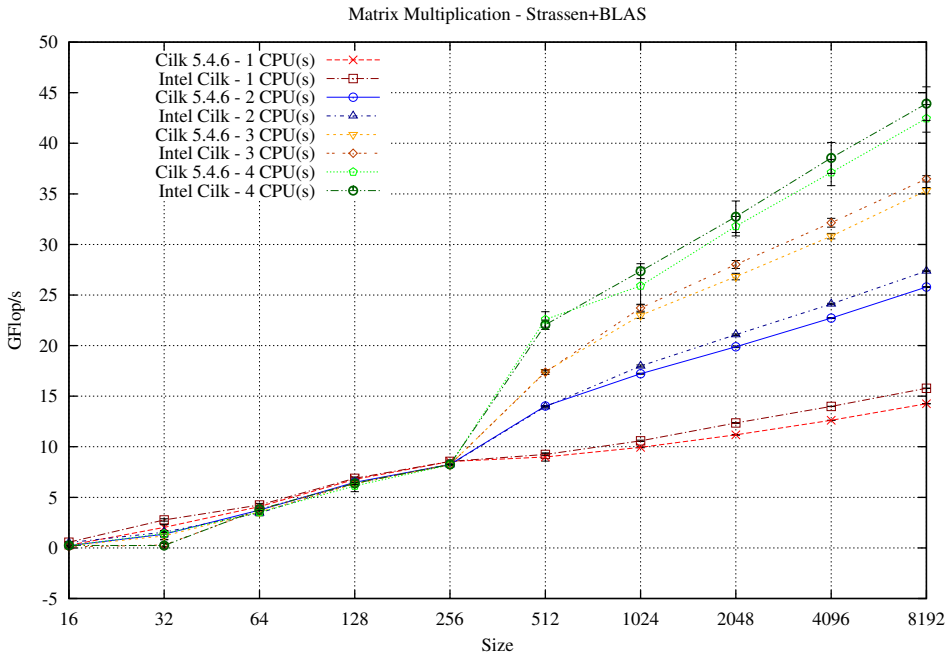


Figura A.3: Comparação do desempenho em GFlops do Cilk 5.4.6 com o Intel Cilk Plus para o algoritmo de Strassen utilizando chamadas à biblioteca BLAS

Ordem da Matriz (n)	**Diferença em GFlops**	**Diferença em %**
16	0,01520199	6,6265683895
32	-1,16293379	-82,905392632
64	0,29467437	8,3655051463
128	0,24848649	4,0559304733
256	0,03898693	0,4748844721
512	-0,52234728	-2,3132502753
1024	1,44295175	5,5683735809
2048	0,93777666	2,9487919088
4096	1,43838952	3,8757673113
8192	1,46003864	3,438469765

Tabela A.2: Diferença no desempenho de pico entre o Cilk 5.4.6 e o Intel Cilk Plus utilizando 4 CPUs para executar o algoritmo de multiplicação de matrizes de Strassen. A primeira coluna indica a ordem das matrizes multiplicadas. A coluna **Diferença em GFlops** apresenta a diferença entre o desempenho do Intel Cilk Plus e o Cilk 5.4.6. A coluna **Diferença em %** mostra em % a diferença de desempenho entre o Intel Cilk Plus e o Cilk 5.4.6.

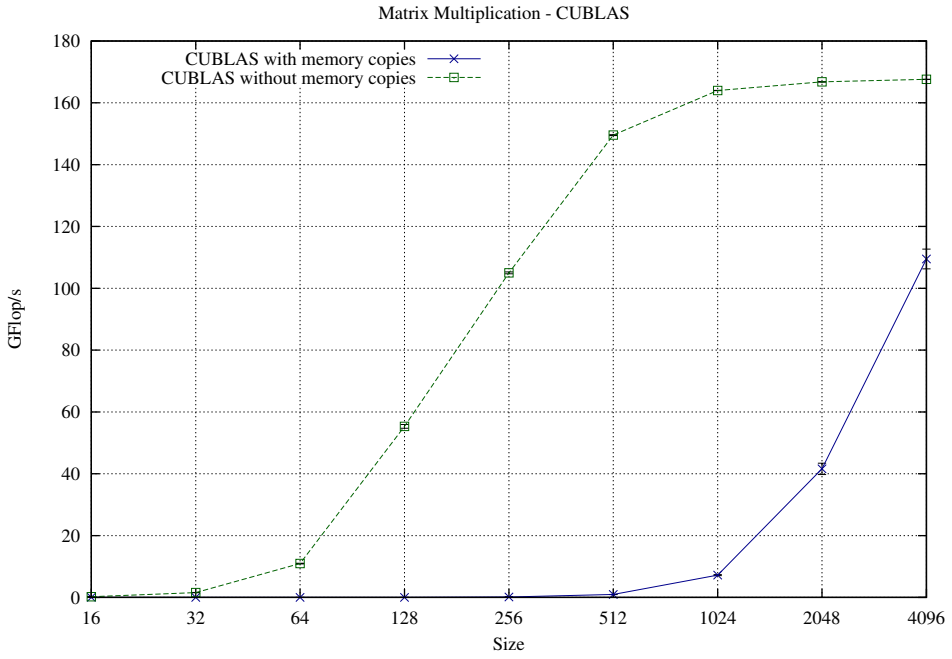


Figura A.4: Comparação do desempenho em GFlops do CUBLAS considerando ou não as transferências de memória.

gasto nas transferências de memória entre a CPU e a GPU. O gráfico da Figura A.4 ilustra o desempenho obtido para matrizes de diferentes tamanhos.

A.3 Experimentos com o *middleware* WORMS

A.3.1 Comparação do desempenho entre as implementações utilizando uma e duas filas

Conforme apresentado na seção 3.5 do Capítulo 3, a implementação atual do WORMS, que utiliza uma fila por *TaskProcessor*, minimizou o sobrecusto ao gerenciar um número elevado de tarefas.

O experimento desta subseção utilizou um algoritmo recursivo para cálculo do i-ésimo número da sequência de Fibonacci apresentado na subseção 2.4.1 do Capítulo 2. A carga de trabalho computacional associada a cada tarefa nesse algoritmo é bastante pequena. Além disso, foram utilizados como critérios de parada para a criação paralela de tarefas os valores de *threshold* 1 e 0 o que provoca a criação de um número de tarefas de 2^i, onde i é o número da sequência de Fibonacci a ser calculado. Com isso, o tempo de execução deste algoritmo é bastante influenciado pelo custo de gerenciamento das tarefas.

A Figura A.5 apresenta o gráfico do tempo de execução do algoritmo recursivo para cálculo do i-ésimo número de Fibonacci nas duas versões do WORMS: a versão inicial implementada com duas filas e a versão atual implementada com apenas uma fila. As execuções das duas versões foram feitas utilizaram 4 *TaskProcessors*, para o maior tamanho testado o número de tarefas criadas é da ordem de 2^{31}. A análise desse gráfico mostra a redução no sobrecusto de gerenciamento de um número elevado de tarefas na versão atual do WORMS em relação a versão inicial.

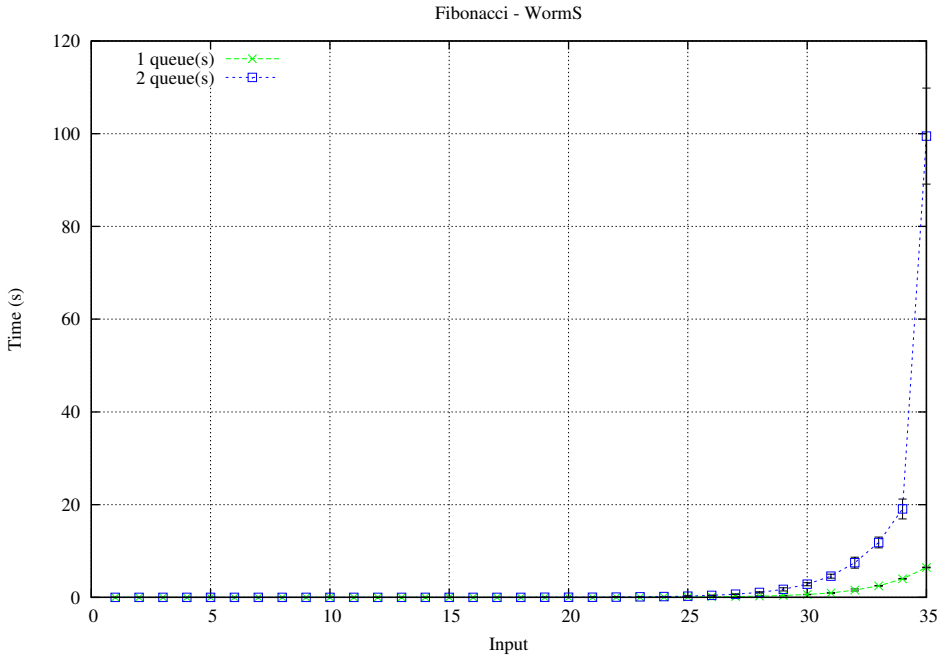


Figura A.5: Comparação do sobrecusto do gerenciamento das tarefas nas duas versões do WORMS.

REFERÊNCIAS

AGRAWAL, K. et al. Adaptive work-stealing with parallelism feedback. **ACM Transactions on Computer Systems**, [S.l.], v.26, n.3, p.1–32, Sept. 2008.

AMD. **AMD Core Math Library (ACML)**. Disponível em http://developer.amd.com/tools/cpu-development/amd-core-math-library-acml/. Acesso em Novembro de 2012.

ASANOVIC, K. et al. A view of the parallel computing landscape. **Commun. ACM**, New York, NY, USA, v.52, p.56–67, Oct. 2009.

ATLAS. **Automatically Tuned Linear Algebra Software (ATLAS)**. Disponível em http://math-atlas.sourceforge.net/. Acesso em Novembro de 2012.

AUGONNET, C. et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: INTERNATIONAL EURO-PAR CONFERENCE ON PARALLEL PROCESSING, 15., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2009. p.863–874. (Euro-Par '09).

AUGONNET, C.; NAMYST, R. A Unified Runtime System for Heterogeneous Multicore Architectures. In: CÉSAR, E. et al. (Ed.). **Euro-Par 2008 Workshops - Parallel Processing**. Berlin, Heidelberg: Springer-Verlag, 2009. p.174–183.

AYGUADE, E. et al. The Design of OpenMP Tasks. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.20, n.3, p.404–418, 2009.

BENDER, M. A.; RABIN, M. O. Scheduling Cilk multithreaded parallel programs on processors of different speeds. In: ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, New York, NY, USA. **Proceedings...** ACM, 2000. p.13–21. (SPAA '00).

BLECHMANN, T. **Boost C++ Libraries - Chapter 17. Boost.Lockfree**. Disponível em http://www.boost.org/doc/libs/1_53_0/doc/html/lockfree.html. Acesso em Fevereiro de 2013.

BLUMOFE, R. D. et al. Cilk: an efficient multithreaded runtime system. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, New York, NY, USA. **Proceedings...** ACM, 1995. p.207–216. (PPOPP '95).

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. In: ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 35., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1994. p.356–368. (SFCS '94).

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. **Journal of the ACM**, [S.l.], v.46, n.5, p.720–748, Sept. 1999.

BOOST. **Boost C++ Libraries**. Disponível em http://www.boost.org/. Acesso em Novembro de 2012.

BORKAR, S.; CHIEN, A. A. The future of microprocessors. **Commun. ACM**, New York, NY, USA, v.54, n.5, p.67–77, May 2011.

BUTTARI, A. et al. The Impact of Multicore on Math Software. In: KÅ GSTRöM, B. et al. (Ed.). **Applied Parallel Computing. State of the Art in Scientific Computing**. [S.l.]: Springer Berlin Heidelberg, 2007. p.1–10. (Lecture Notes in Computer Science, v.4699).

CALLAHAN, D. Design Considerations For Parallel Programming. **MSDN magazine**, [S.l.], v.23, n.11, p.74 – 85, Oct. 2008.

CEDERMAN, D.; TSIGAS, P. On Dynamic Load Balancing on Graphics Processors. **Technology**, [S.l.], p.57–64, 2008.

CEDERMAN, D.; TSIGAS, P. Dynamic Load Balancing Using Work-Stealing. In: HWU, W.-M. W. (Ed.). **GPU Computing Gems**: jade edition. [S.l.]: Elsevier, 2011. p.485–499.

CHAPMAN, B.; JOST, G.; PAS, R. **Using OpenMP**: portable shared memory parallel programming. [S.l.]: MIT Press, 2007. n.v. 10. (Scientific and engineering computation).

CORMEN, T. H. et al. **Introduction to Algorithms, Third Edition**. [S.l.]: MIT Press, 2009.

CORPORATION, I. **Intel Threading Building Blocks (Intel TBB)**. Disponível em http://threadingbuildingblocks.org/. Acesso em Novembro de 2012.

CORPORATION, I. **Cilk Home Page | CilkPlus**. Disponível em http://www.cilkplus.org/. Acesso em Janeiro de 2013.

DIJKSTRA, E. W. Solution of a problem in concurrent programming control. **Commun. ACM**, New York, NY, USA, v.8, n.9, p.569–, Sept. 1965.

DINAN, J. et al. Scalable work stealing. In: CONFERENCE ON HIGH PERFORMANCE COMPUTING NETWORKING, STORAGE AND ANALYSIS - SC - SUPERCOMPUTING '09, New York, New York, USA. **Proceedings...** ACM Press, 2009. p.1.

DUFFY, D. An Introduction to Thread. In: DEMMING, R.; DUFFY, D. (Ed.). **Introduction to the Boost C++ Libraries; Volume I - Foundation**. [S.l.]: Datasim Education Bv, 2010. n.1.

FORUM, M. P. I. **MPI-2: extensions to the message-passing interface**. Knoxville, TN, USA: University of Tennessee, 1997.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The implementation of the Cilk-5 multithreaded language. In: ACM SIGPLAN 1998 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION - PLDI '98, New York, New York, USA. **Proceedings...** ACM Press, 1998. p.212–223.

GAUTIER, T.; BESSERON, X.; PIGEON, L. KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PARALLEL SYMBOLIC COMPUTATION, 2007., New York, NY, USA. **Proceedings...** ACM, 2007. p.15–23. (PASCO '07).

GRAMA, A. **Introduction to Parallel Computing**. [S.l.]: Addison-Wesley, 2003. (Pearson Education).

GROUP, M. C. S. R. **The Cilk Project**. Disponível em http://supertech.csail.mit.edu/cilk/. Acesso em Novembro de 2012.

GUO, Y. et al. Work-first and help-first scheduling policies for async-finish task parallelism. **2009 IEEE International Symposium on Parallel Distributed Processing**, [S.l.], v.0, p.1–12, 2009.

HERMANN, E. et al. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In: EURO-PAR CONFERENCE ON PARALLEL PROCESSING: PART II, 16., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2010. p.235–246. (Euro-Par'10).

HWU, W.-M. W. Thrust - A Productivity-Oriented Library for CUDA. In: **GPU Computing Gems**: jade edition. [S.l.]: Elsevier, 2011. p.359–373.

INRIA; MOAIS; LIG. **XKAAPI - Kernel for Adaptative, Asynchronous Parallel and Interactive programming**. Disponível em http://kaapi.gforge.inria.fr/. Acesso em Novembro de 2011.

KAMBADUR, P. et al. PFunc: modern task parallelism for modern high performance computing. In: CONFERENCE ON HIGH PERFORMANCE COMPUTING NETWORKING, STORAGE AND ANALYSIS - SC - SUPERCOMPUTING '09, New York, New York, USA. **Proceedings...** ACM Press, 2009. p.1.

KISTLER, M. et al. Petascale Computing with Accelerators. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 14., New York, NY, USA. **Proceedings...** ACM, 2009. p.241–250. (PPoPP '09).

LARUS, J. Spending Moore's dividend. **Commun. ACM**, New York, NY, USA, v.52, n.5, p.62–69, May 2009.

LEISERSON, C. E. The Cilk++ concurrency platform. In: ANNUAL DESIGN AUTOMATION CONFERENCE, 46. **Proceedings...** ACM, 2009. p.522–527.

LIMA, J. V. F. et al. Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD), 24., Columbia University, New York, USA. **Anais...** [S.l.: s.n.], 2012.

LING, B. **The Boost C++ Libraries**. [S.l.]: XML Press, 2011.

MATTSON, T.; SANDERS, B.; MASSINGILL, B. **Patterns for parallel programming**. [S.l.]: Addison-Wesley, 2005. (Software patterns series).

MICHAEL, M. M.; VECHEV, M. T.; SARASWAT, V. A. Idempotent work stealing. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 14., New York, NY, USA. **Proceedings...** ACM, 2009. p.45–54. (PPoPP '09).

MOR, S. D. K. **Escalonamento on-line eficiente de programas fork-join recursivos do tipo divisão e conquista em MPI**. 2010. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação.

MOR, S.; MAILLARD, N. Dynamic workload balancing deques for branch and bound algorithms in the message passing interface. **Int. J. High Perform. Syst. Archit.**, Inderscience Publishers, Geneva, SWITZERLAND, v.3, n.2/3, p.77–86, May 2011.

NAVAUX, P.; ROSE, C. **ARQUITETURAS PARALELAS**. Porto Alegre: BOOKMAN COMPANHIA ED, 2008.

NEAPOLITAN, R.; NAIMIPOUR, K. **Foundations of Algorithms**. [S.l.]: Jones & Bartlett Learning, 2010.

NVIDIA. **CUDA C Programming Guide**. [S.l.: s.n.], 2011.

NVIDIA Developer Zone. **CUDA Libraries**. Disponível em https://developer.nvidia.com/technologies/Libraries. Acesso em Novembro de 2012.

NVIDIA Developer Zone. **NVIDIA CUDA Basic Linear Algebra Subroutines**. Disponível em https://developer.nvidia.com/cublas. Acesso em Novembro de 2012.

Open MPI. **Portable Hardware Locality (hwloc)**. Disponível em http://www.open-mpi.org/projects/hwloc/. Acesso em Fevereiro de 2013.

OpenMP ARB. **OpenMP Application Program Interface v3.0**. [S.l.]: OpenMP Architecture Review Board, 2008.

PEREZ, J. M.; BADIA, R. M.; LABARTA, J. A dependency-aware task-based programming environment for multi-core architectures. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 2008. **Anais...** IEEE, 2008. p.142–151.

PINTO, V. G. et al. Escalonamento baseado em roubo de tarefas em arquiteturas paralelas híbridas. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, 12., Porto Alegre. **Anais...** Sociedade Brasileira de Computação, 2012. p.89–90. (Anais da Escola Regional de Desempenho, v.1).

PINTO, V. G.; MAILLARD, N. Work Stealing on Hybrid Architectures. In: SYMPOSIUM ON COMPUTER SYSTEMS (WSCAD-SSC 2012), 13., Los Alamitos. **Anais...** IEEE Computer Society, 2012. p.17 –24.

PINTO, V. G.; MAILLARD, N. Scheduling by Work-Stealing in Hybrid Parallel Architectures. In: X WORKSHOP DE PROCESSAMENTO PARALELO E DISTRIBUíDO, Porto Alegre. **Anais...** [S.l.: s.n.], 2012. p.55–58.

RAUBER, T.; RÜNGER, G. **Parallel Programming**: for multicore and cluster systems. [S.l.]: Springer, 2010.

REINDERS, J. **Intel Threading Building Blocks**: outfitting c++ for multi-core processor parallelism. [S.l.]: O'Reilly Media, 2010. (O'Reilly Series).

SEBESTA, R. **Conceitos de Linguagens de Programacao**. [S.l.]: BOOKMAN COMPANHIA ED, 2003.

TOMOV, S.; DONGARRA, J.; BABOULIN, M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. **Parallel Comput.**, Amsterdam, The Netherlands, The Netherlands, v.36, n.5-6, p.232–240, June 2010.

TOP500. **TOP500 Supercomputer Sites**. Disponível em http://www.top500.org/. Acesso em Novembro de 2012.

TOSS, J.; GAUTIER, T. A New Programming Paradigm for GPGPU. In: KAKLAMANIS, C.; PAPATHEODOROU, T.; SPIRAKIS, P. (Ed.). **Euro-Par 2012 Parallel Processing**. [S.l.]: Springer Berlin Heidelberg, 2012. p.895–907. (Lecture Notes in Computer Science, v.7484).

VALIANT, L. G. A bridging model for parallel computation. **Commun. ACM**, New York, NY, USA, v.33, n.8, p.103–111, Aug. 1990.

VANDIERENDONCK, H.; PRATIKAKIS, P.; NIKOLOPOULOS, D. S. Parallel programming of general-purpose programs using task-based programming models. In: USENIX CONFERENCE ON HOT TOPIC IN PARALLELISM, 3., Berkeley, CA, USA. **Proceedings...** USENIX Association, 2011. p.13. (HotPar'11).

WATT, D. **Programming Language Design Concepts**. [S.l.]: Wiley, 2004.

WHALEY, R. C.; DONGARRA, J. J. Automatically tuned linear algebra software. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING (CDROM), 1998., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1998. p.1–27. (Supercomputing '98).

WILLIAMS, A. **Boost C++ Libraries - Chapter 28. Thread**. Disponível em http://www.boost.org/doc/libs/1_49_0/doc/html/thread.html. Acesso em Novembro de 2012.